华章IT

容器技术系列

Docker源码分析

THE SOURCE CODE ANALYSIS OF DOCKER

孙宏亮 著

- 国内首部Docker源码分析著作
- 从源码角度全面解析Docker设计与实现
- 填补Docker理论与实践之间的鸿沟

机械工业出版社
China Machine Press

容器技术系列

Docker源码分析

孙宏亮　著

ISBN：978-7-111-51072-7

本书纸版由机械工业出版社于2015年出版，电子版由华章分社（北京华章图文信息有限公司，北京奥维博世图书发行有限公司）全球范围内制作与发行。

版权所有，侵权必究

客服热线：+ 86-10-68995265

客服信箱：service@bbbvip.com 官方网址：www.hzmedia.com.cn 新浪微博 @华章数媒

腾讯微博 @yanfabook 微信公众号 华章电子书（微信号：hzebook）

目录

赞誉

序

前言

第1章　Docker架构

1.1　引言

1.2　Docker总架构图

1.3　Docker各模块功能与实现分析

1.3.1　Docker　Client

1.3.2　Docker　Daemon

1.3.3　Docker　Registry

1.3.4　Graph

1.3.5　Driver

1.3.6　libcontainer

1.3.7　Docker　Container

1.4　Docker运行案例分析

1.4.1　docker　pull

1.4.2　docker　run

1.5　总结

第2章　Docker　Client创建与命令执行

2.1　引言
2.2　创建Docker　Client
2.2.1　Docker命令的flag参数解析
2.2.2　处理flag信息并收集Docker　Client的配置信息
2.2.3　如何创建Docker　Client
2.3　Docker命令执行
2.3.1　Docker　Client解析请求命令
2.3.2　Docker　Client执行请求命令
2.4　总结
第3章　启动Docker　Daemon
3.1　引言
3.2　Docker　Daemon的启动流程
3.3　mainDaemon（　）的具体实现
3.3.1　配置初始化
3.3.2　flag参数检查
3.3.3　创建engine对象
3.3.4　设置engine的信号捕获
3.3.5　加载builtins
3.3.6　使用goroutine加载daemon对象并运行
3.3.7　打印Docker版本及驱动信息
3.3.8　serveapi的创建与运行

3.4　总结
第4章　Docker　Daemon之NewDaemon实现
4.1　引言
4.2　NewDaemon具体实现
4.3　应用配置信息
4.3.1　配置Docker容器的MTU
4.3.2　检测网桥配置信息
4.3.3　查验容器间的通信配置
4.3.4　处理网络功能配置
4.3.5　处理PID文件配置
4.4　检测系统支持及用户权限
4.5　配置工作路径
4.6　加载并配置graphdriver
4.6.1　创建graphdriver
4.6.2　验证btrfs与SELinux的兼容性
4.6.3　创建容器仓库目录
4.6.4　迁移容器至aufs类型
4.6.5　创建镜像graph
4.6.6　创建volumesdriver以及volumes　graph
4.6.7　创建TagStore
4.7　配置Docker　Daemon网络环境

4.7.1　创建Docker网络设备
4.7.2　启用iptables功能
4.7.3　启用系统数据包转发功能
4.7.4　创建DOCKER链
4.7.5　注册处理方法至Engine
4.8　创建graphdb并初始化
4.9　创建execdriver
4.10　创建daemon实例
4.11　检测DNS配置
4.12　启动时加载已有Docker容器
4.13　设置shutdown的处理方法
4.14　返回daemon对象实例
4.15　总结
第5章　Docker　Server的创建
5.1　引言
5.2　Docker　Server创建流程
5.2.1　创建名为“serveapi”的Job
5.2.2　配置Job环境变量
5.2.3　运行Job
5.3　ServeApi运行流程
5.4　ListenAndServe实现

5.4.1　创建router路由实例
5.4.2　创建listener监听实例
5.4.3　创建http.Server
5.4.4　启动API服务
5.5　总结
第6章　Docker　Daemon网络
6.1　引言
6.2　Docker　Daemon网络介绍
6.3　Docker　Daemon网络配置接口
6.4　Docker　Daemon网络初始化
6.4.1　启动Docker　Daemon传递flag参数
6.4.2　解析网络flag参数
6.4.3　预处理flag参数
6.4.4　确定Docker网络模式
6.5　创建Docker网桥
6.5.1　提取环境变量
6.5.2　确定Docker网桥设备名
6.5.3　查找bridgeIface网桥设备
6.5.4　bridgeIface已创建
6.5.5　bridgeIface未创建
6.5.6　获取网桥设备的网络地址

6.5.7　配置Docker　Daemon的iptables
6.5.8　配置网络设备间数据报转发功能
6.5.9　注册网络Handler
6.6　总结
第7章　Docker容器网络
7.1　引言
7.2　Docker容器网络模式
7.2.1　bridge桥接模式
7.2.2　host模式
7.2.3　other　container模式
7.2.4　none模式
7.3　Docker　Client配置容器网络模式
7.3.1　使用Docker　Client
7.3.2　runconfig包解析
7.3.3　CmdRun执行
7.4　Docker　Daemon创建容器网络流程
7.4.1　创建容器之网络配置
7.4.2　启动容器之网络配置
7.5　execdriver网络执行流程
7.5.1　创建libcontainer的Config对象
7.5.2　调用libcontainer的namespaces启动容器

7.6　libcontainer实现内核态网络配置
7.6.1　创建exec.Cmd
7.6.2　启动exec.Cmd创建进程
7.6.3　为容器进程初始化网络环境
7.7　总结
第8章　Docker镜像
8.1　引言
8.2　Docker镜像介绍
8.3　rootfs
8.4　Union　Mount
8.5　image
8.6　layer
8.7　总结
第9章　Docker镜像下载
9.1　引言
9.2　Docker镜像下载流程
9.3　Docker　Client
9.3.1　解析镜像参数
9.3.2　配置认证信息
9.3.3　发送API请求
9.4　Docker　Server

9.4.1 解析请求参数
9.4.2 创建并配置Job
9.4.3 触发执行Job
9.5 Docker Daemon
9.5.1 解析Job参数
9.5.2 创建session对象
9.5.3 执行镜像下载
9.6 总结
第10章 Docker镜像存储
10.1 引言
10.2 镜像注册
10.3 验证镜像ID
10.4 创建镜像路径
10.4.1 创建mnt、diff和layers子目录
10.4.2 挂载祖先镜像并返回根目录
10.5 存储镜像内容
10.5.1 解压镜像内容
10.5.2 收集镜像大小并记录
10.5.3 存储jsonData信息
10.6 注册镜像ID
10.7 总结

第11章　docker　build实现
11.1　引言
11.2　docker　build执行流程
11.2.1　Docker　Client与docker　build
11.2.2　Docker　Server与docker　build
11.2.3　Docker　Daemon与docker　build
11.3　Dockerfile命令解析流程
11.4　Dockerfile命令分析
11.4.1　FROM命令
11.4.2　RUN命令
11.4.3　ENV命令
11.5　总结
第12章　Docker容器创建
12.1　引言
12.2　Docker容器运行流程
12.3　Docker　Daemon创建容器对象
12.3.1　LookupImage
12.3.2　CheckDepth
12.3.3　mergeAndVerifyConfig
12.3.4　newContainer
12.3.5　createRootfs

12.3.6　ToDisk
12.3.7　Register
12.4　Docker　Daemon启动容器
12.4.1　setupContainerDns
12.4.2　Mount
12.4.3　initializeNetworking
12.4.4　verifyDaemonSetting
12.4.5　prepareVolumesForContainer
12.4.6　setupLinkedContainers
12.4.7　setupWorkingDirectory
12.4.8　createDaemonEnvironment
12.4.9　populateCommand
12.4.10　setupMountsForContainer
12.4.11　waitForStart
12.5　总结
第13章　dockerinit启动
13.1　引言
13.2　dockerinit介绍
13.2.1　dockerinit初始化内容
13.2.2　dockerinit与Docker　Daemon
13.3　dockerinit执行入口

13.3.1　createCommand分析
13.3.2　namespace.exec
13.4　dockerinit运行
13.4.1　reexec.Init（　）的分析
13.4.2　dockerinit的执行流程
13.5　libcontainer的运行
13.5.1　Docker　Daemon设置cgroups参数
13.5.2　Docker　Daemon创建网络栈资源
13.5.3　dockerinit配置网络栈
13.5.4　dockerinit初始化mount　namespace
13.5.5　dockerinit完成namespace配置
13.5.6　dockerinit执行用户命令Entrypoint
13.6　总结
第14章　libcontainer介绍
14.1　引言
14.2　Docker、libcontainer以及LXC的关系
14.3　libcontainer模块分析
14.3.1　namespace
14.3.2　cgroup
14.3.3　网络
14.3.4　挂载

14.3.5　设备
14.3.6　nsinit
14.3.7　其他模块
14.4　总结
第15章　Swarm架构设计与实现
15.1　引言
15.2　Swarm架构
15.2.1　Swarm　Node
15.2.2　Docker　Node
15.2.3　node　discovery
15.2.4　scheduler
15.3　Swarm命令
15.3.1　swarm　create
15.3.2　swarm　manage
15.3.3　swarm　join
15.3.4　swarm　list
15.4　总结
第16章　Machine架构设计与实现
16.1　引言
16.2　Machine架构
16.2.1　Machine

16.2.2　Store

16.2.3　Host

16.2.4　Driver

16.2.5　Provisioner

16.2.6　Machine运行流程

16.3　Machine与Swarm的结合

16.4　总结

第17章　Compose架构设计与实现

17.1　引言

17.2　Compose介绍

17.3　Compose架构

17.4　Compose评价

17.4.1　Compose单机能力

17.4.2　Compose跨节点能力

17.4.3　Compose与Swarm

17.5　总结

赞誉

像谷歌一样部署自己的应用，这是很多软件工程师的梦想。Docker的目标是圆很多人的梦。自从InfoQ推出Docker系列文章，作为操作系统课程教师的我一直在学习并关注Docker的茁壮成长。

当我发现这上面刊登的“Docker源码分析”系列文章的作者居然是我们课程组的研究生助教孙宏亮时，惊喜之情溢于言表。宏亮对Docker的理解十分深刻，他本人是Docker的积极拥护者、倡导者和贡献者。他在研究生毕业以后投身到了创业公司DaoCloud，去为Docker的梦想开创美好的未来。

最近，我又欣喜地发现，这系列文章以及后续章节即将正式出版成书，有机会同更多的Docker用户、开发者、学习者见面。本书通过分析解读Docker源码，让读者了解Docker的内部结构和实现，以便更好地使用Docker。该书的内容组织深入浅出，表述准确到位，有大量流程图和代码片段帮助读者理解Docker各个功能模块的流程，是学习Docker开源系统的良师益友。

——寿黎旦，浙江大学计算机学院教授

近年来，Docker迅速风靡了云计算世界，但是专门针对Docker的技术实现进行深入分析的文章却相对较少。这一方面由于Docker技

术变化很快，源码分析很快会跟不上版本发展；另一方面，对源代码的解析，需要对整个Docker设计具备全局的视角，才能深入浅出地找到源码中的脉络。

宏亮的这本《 Docker源码分析》，恰如其时的出现，弥补了这个空白，对于希望参与到Docker社区、参与代码贡献或构建自己的Docker应用环境的读者来说，应是一本案头必备书籍。

——王兴宇，《 Linux中国》创始人

在崇尚源码至上的工程师文化里，文档介绍、发布会材料都是苍白的，唯有研读源码，才能深刻理解软件背后的原理。与所有其他软件一样，读源码并不是学习Docker最快的途径，但是如果有人通读源码后给出了详细分析，你就可以轻松地站在巨人的肩膀上。

很高兴看到国内这么快就出版源码分析的书籍。对于所有想在Docker方面进阶和想晋升为高端用户的读者，都值得阅读本书，也希望通过《 Docker源码分析》一书，可以诞生更多的社区贡献者，共同推动Docker的发展。

——黄强，华为Docker Committer

值得自豪的是，就在Docker蓬勃发展之际，第一本详尽剖析Docker源码的著作出自国人之手。

本书在每章宏观的流程梳理背后都伴随有更加细致深入的源码分析。无论读者是只想了解使用Docker，还是抱着深入理解Docker并参与社区开发、二次开发的心态，本书都值得一读。

——胡科平，华为Docker Committer

这本书从源码的角度对Docker的实现原理进行了深入的探讨和细腻的讲解，将当前热门的容器技术的背后机理讲解得深入浅出明白透彻。无论是Docker的用户还是开发者，通过阅读本书都可以对Docker有更深刻的理解，能够更好地使用或者开发Docker。

——雷继棠，华为Docker Committer

Docker已经是一个成长2年时间的云计算技术，它正在以惊人的速度在全球范围内扩张自己的“疆土”。我作为Docker中国区的开发者，非常希望能看到有一本书详细地告诉我，Docker的每一个细节是如何实现的。所以当我在InfoQ上看到宏亮的“Docker源码分析”专栏时眼前一亮。今天，它终于汇编成书摆在你我的眼前。我希望你能在这本书上学到更多Docker技术的精髓思想，在实战Docker技术时可以运用自如!

——肖德时，数人科技CTO

我家里的书柜中至今仍然保留着一本《 Linux内核完全注释》，它伴随我很长一段时间，使我受益匪浅。10年之后，当我拿到宏亮的《 Docker源代码分析》草稿，昨日又重现。这本书无论是对学习Docker，还是掌握Go语言，都是非常好的一手资源。雷锋不常有，大家要支持!

——赵鹏，VisualOps创始人

序

三年前，我在VMware负责Cloud Foundry这一款开源PaaS产品在中国的开发者社区和生态系统建设工作。当时关于Cloud Foundry的中文文档非常少，更不用说有深度的技术干货了，所以当CSDN上出现了一个专门从底层模块和源码的角度，对Cloud Foundry做深度剖析的系列博客文章时，一下子就引起了我的注意。

经过一番“人肉搜索”，我非常惊奇的发现，这一系列干货的作者孙宏亮，竟然是浙江大学计算系的一年级研究生。当时，VMware跟浙江大学在Cloud Foundry方面有比较深入的合作，浙大计算机系的SEL实验室，投入了精锐的师资力量，从事分布式系统和新一代PaaS的研究工作。宏亮初入浙江大学，就在这样的氛围下开始了他的研究生求学生涯，应当说是非常幸运的。

宏亮在CSDN上的系列文章，主打“源码分析”，这正是当时开源社区内比较缺少的内容。提笔写“源码分析”，需要一定的勇气，阅读源码需要耗费大量的精力，需要从数十万行代码中整理出清晰的逻辑，从中抽丝拨茧、概括精华，这是一份非常辛苦的工作。而且，分析源码也需要一些“挑战权威”的精神，不仅仅是简单的代码解析，更需要提炼出自己的观点，甚至敢于发现和纠正一些已有的问题。

在源码分析方面，宏亮充分体现了“初生牛犊不怕虎”的精神，非常详细地剖析了当时Cloud Foundry的几个主要模块，条理清晰，技术分析准确到位。宏亮这一系列文章帮助了包括一线互联网公司在内的许多企业了解、认识和最终使用Cloud Foundry，宏亮也借此奠定了他在PaaS社区的“江湖地位”。

从去年开始，Docker的热潮开始波及中国开发者社区。我有幸跟宏亮一起在CSDN主办的第一届Container技术大会上发表联合主题演讲，向来宾介绍Cloud Foundry内部的容器技术实现，以及对Docker的一些展望。那次大会是一个很重要的里程碑，在那之后，宏亮开始深入研究Docker的底层实现，并在InfoQ连载“Docker源码分析”系列文章。

对PaaS平台的研究越深入，越能够发现和体会Docker对开发和运维的价值。如果说当初的Cloud Foundry模块和源码分析文章，是读研期间的学习笔记，那这次宏亮的《Docker源码分析》，则是一个经过了深思熟虑、系统性、结构化的工作。Docker开源项目发展速度非常快，这次在文章连载内容的基础上出书，为了保证内容的准确性和时效性，宏亮补充了大量Docker最新项目的内容，特别是对Swarm、Machine和Compose这三个模块的开发进展做了紧密的追踪。

这是一本从架构和代码角度讲解Docker底层实现的技术图书，我从连载第一篇开始就对这个系列的文章保持了紧密的关注，也目睹了宏亮在后期整理加工成书过程中的辛勤努力。在《 Docker源码分析》成书付梓出版之际，非常幸运，能够为宏亮写着一篇推荐序。这本书非常适合以下三类读者学习和阅读。

·希望以Docker容器交付软件的程序员。

Docker是未来互联网软件的交付件，这件事随着OCP标准的制定，正在逐渐成为事实。程序员和运维工程师都需要了解Docker的工作方式，尤其是Docker镜像的结构，软件通过Dockerfile打包时的优化方式等，这些内容在本书中都有非常详细的阐述。

·Docker化云计算平台的建设者和维护者。

Docker公司在今年的全球开发者大会上提出了“Production Ready”的口号，有越来越多的互联网公司和传统企业采用Docker来构建开发、测试和运维平台。Docker在网络、存储、安全等领域的细节，是平台建设者和维护者必须深入了解的部分，这些领域还在不断变化，新的项目 也层出不穷，但本书对网络、存储和安全的基本知识和概念，做了非常清晰的介绍，也深入到了底层实现的代码。

·Go语言程序员。

即使不在项目中使用Docker，本书也能够为Go语言程序员带来帮助。Docker项目中大量采用了Go语言，尤其是在处理并发场景时，Docker对Go语言的运用可谓出神入化。本书可以帮助Go语言程序员亲身体验特大型项目中Go语言的威力，以及实战场景中Golang模式和功能的用法。

最后，预祝宏亮在Docker的学习和工作中再创佳绩，也希望读者可以从本书收获知识，开阔眼界。

喻勇

2015年7月13日

前言

Docker是什么

Docker从2013年诞生，短短两年时间就在全球IT技术圈内迅速走红，实乃技术圈内不可忽视的一阵飓风。然而，Docker是什么，Docker带来了什么？

Docker官方如此描述Docker：“Build，Ship，Run.An open platform for distributed applications for developers and sysadmins”。换言之，Docker为开发者与系统管理者提供了分布式应用的开放平台，从而可以便捷地构建、迁移与运行分布式应用。

多年来，IT行业中开发与运维一直是两个界限清晰的词。开发工程师专门从事软件的开发工作，最终交付软件代码；运维工程师则部署前者交付的软件，并接管软件的运行与管理。然而，在长时间的实践过程中，开发与运维分离的方式难免存在弊病，两者职责的过分清晰导致软件效率的降低。随着分布式系统的流行，系统规模越来越大，软件越来越复杂，系统环境配置暴露的问题层出不穷。究其缘由，还是因为开发人员缺少软件运行环境的认知，而运维人员对软件逻辑所知甚少。在这样的背景下，DevOps横空出世，提倡开发与运维不可分割，协调并进。

Docker无疑是DevOps大潮中最具实践价值的不二法宝。Docker从Linux内核的角度出发，属于轻量级虚拟化技术，有能力秒级提供应用隔离环境，完成云计算时代分布式应用的第一需求“隔离”。另外，Docker的镜像技术利用联合文件系统的优势，自下至上打包系统软件、系统环境以及软件程序，将运行环境与应用程序灵活地结合，快速运行Docker化的应用程序。同时，可读性极强的Dockerfile，极大地简化镜像的复杂性，并为镜像的转移与重新构建提供了可能性。

Docker提供轻便的资源分配方式，解决应用运行与系统环境的依赖，弥合应用跨节点迁移的鸿沟，种种特性都表明Docker几乎就是为“云计算”而生的。如今，Docker社区不断扩大并健康发展，多家国际IT巨头也纷纷宣布支持Docker，这一切更是让全球IT人士对Docker的未来充满信心。

本书的内容

本书是一本引导读者了解Docker实现原理的技术普及书。笔者一直坚信，了解软件或者系统最直接、最透彻的方式就是研读它们的源码。“源码即文档”也是鼓励开发者能更多地从源码的角度去学习软件或系统的本质。

本书的内容主要集中于3个部分：Docker的架构，Docker的模块，Docker的三驾马车Swarm、Machine以及Compose。

第一部分，主要从宏观的角度和读者一起领略Docker的架构设计，并初步介绍架构中各模块的职责。

第二部分是本书的主体部分，主要针对Docker中多个重要的模块进行具体深入分析，包括：Docker Client、Docker Daemon、Docker Server、Docker网络、Docker镜像、Docker容器等。读者可以发现，Docker的模块之间耦合度非常低，很适合循序渐进，层层深入。第2章至第8章主要从Docker软件的架构入手，勾勒骨架；第9章至第11章重点讨论Docker镜像技术，夯实基础；第12章至第14章则进一步分析Docker容器的始末，阐述本质。

第三部分介绍Docker生态三驾马车Swarm、Machine、Compose。Docker拥有强大的单机能力，三驾马车可以很好地补充Docker的跨主机能力以及部署能力。读者可以通过第15章至第17章感受Docker生态圈中其他功能强大的软件。

希望本书能够让读者最大化地感受Docker的神奇与魅力。

关于勘误

由于时间与水平都比较有限，因此本书难免会存在一些纰漏和错误。如果读者发现了问题，请及时与我联系。我也会在本书后续的版本中加以改正。我的邮箱是：allen.sun@daocloud.io。我非常希望和大家一起学习与讨论Docker，并共同推动Docker在社区的发展。

致谢

最后，向本书编写过程中给予我巨大帮助的人们表示最诚挚的感谢。感谢我的父母，没有他们的鼓励和支持，此书不可能在如此短的时间内完成。感谢我的母校浙江大学以及SEL实验室的老师与同学们，是他们在我求学过程中给予无私的指引与帮助。感谢我的同事熊中祥，是他在本书编写过程中提出了很多宝贵的建议，尤其在Machine和Compose部分。感谢我的同事喻勇和冯钊，他们为本书的编写做了很多沟通与协调工作。最后，还要感谢华章公司的编辑们，她们认真细致的工作，使本书以完美的形式展现给各位读者。

孙宏亮

2015年6月

第1章　Docker架构

1.1　引言

Docker是Linux平台上的一款轻量级虚拟化容器的管理引擎。在全球范围内，Docker还是一个开源项目，整个项目基于Go语言开发，代码托管于GitHub网站上，并遵从Apache 2.0协议。目前，Docker可以帮助用户在容器内部快速自动化部署应用，并利用Linux内核特性命名空间（namespaces）及控制组（cgroups）等为容器提供隔离的运行环境。Docker借助操作系统层的虚拟化实现资源的隔离，因此Docker容器在运行时与虚拟机（VM）的运行有很大的区别，Docker容器与宿主机共享同一个操作系统，不会有额外的操作系统开销。这样的优势很明显，因而大大提高了资源利用率，并且提升了I/O等方面的性能。

众多新颖的特性以及项目本身的开放性，导致Docker在不到两年的时间里迅速获得了诸多厂商的青睐，其中更是包括Google、Microsoft、VMware等行业领航者。Google在2014年6月推出了Kubernetes，宣布支持Docker容器的调度管理，而2014年8月Microsoft则宣布Azure上支持Kubernetes，随后传统虚拟化巨头

VMware宣布与Docker强强合作。2014年9月中旬，Docker更是获得4000万美元的C轮融资，以推动分布式应用的发展。

目前，Docker的前景被普遍看好。未来的云计算领域乃至整个IT领域，Docker都必将扮演不可或缺的角色。为了帮助大家更好地认识Docker、理解Docker并掌握Docker，本书从Docker源码的角度出发，详细介绍Docker的架构、Docker的运行以及Docker的特性。本章主要介绍Docker架构。

本书关于Docker的分析均基于Docker 1.2.0版本的源码。

本章目的是在理解Docker源码的基础上分析Docker架构，分析过程主要按照以下三个部分进行：

·Docker的总架构图。

·Docker架构内部各模块功能与实现的分析。

·通过具体的Docker命令，阐述Docker的运行流程。

1.2　Docker总架构图

作为Linux平台上的一种容器的管理引擎，Docker并不像其他大型分布式系统那样复杂。Docker的源码总量并不多，而且清晰的源码结构使得Docker的学习成本并不高。换言之，Docker源码的学习过程并不枯燥，我们可以从中学到很多东西，如Go语言的运用、Docker架构的设计原理等。Docker对用户而言是一个简单的C/S架构，用户通过客户端与服务器端建立通信，而Docker的后端是一个松耦合的架构，架构中的模块各司其职、有机组合，支撑着Docker运行。

Docker的总架构如图1-1所示。架构中主要的模块有：DockerClient、DockerDaemon、Docker Registry、Graph、Driver、libcontainer以及Docker Container。

对用户而言，Docker Client是与Docker Daemon建立通信的最佳途径。用户通过Docker Client发起容器的管理请求，请求最终发往Docker Daemon。

Docker Daemon作为Docker架构中的主体部分，首先具备服务端的功能，有能力接收Docker Client发起的请求；其次具备

Docker Client请求的处理能力。Docker Daemon内部所有的任务均由Engine来完成，且每一项工作都以一个Job的形式存在。

Docker Daemon需要完成的任务很多，因此Job的种类也很多。若用户需要下载容器镜像，Docker Daemon则会创建一个名为“pull”的Job，运行时从Docker Registry中下载镜像，并通过镜像管理驱动graphdriver将下载的镜像存储在graph中；若用户需要为Docker容器创建网络环境，Docker Daemon则会创建一个名“allocate_interface”的Job，通过网络驱动networkdriver分配网络接口的资源……

libcontainer是一套独立的容器管理解决方案，这套解决方案涉及了大量Linux内核方面的特性，如：namespaces、cgroups以及capabilities等。libcontainer很好地抽象了Linux的内核特性，并提供完整、明确的接口给Docker Daemon。

当用户执行运行容器这个命令之后，一个Docker容器就处于运行状态，该容器拥有隔离的运行环境、独立的网络栈资源以及受限的资源等。

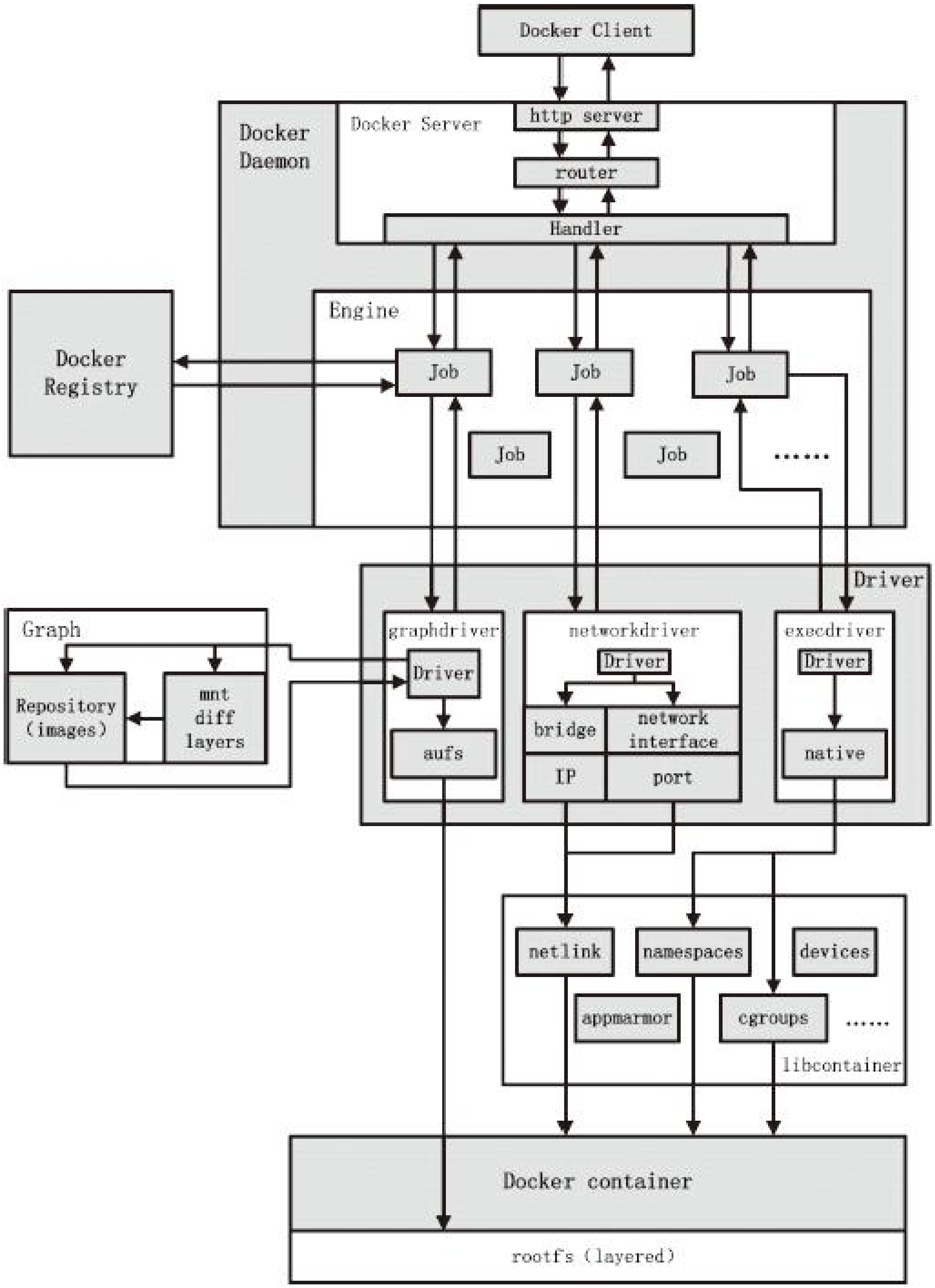


图1-1　Docker总架构图

1.3 Docker各模块功能与实现分析

下面我们将从Docker的总架构图入手，抽离出架构内的各个模块，并对各个模块进行更为细化的架构分析与功能阐述。

1.3.1 Docker Client

Docker Client是Docker架构中用户与Docker Daemon建立通信的客户端。在一台安装有Docker的机器上，用户可以使用可执行文件docker作为Docker Client，发起众多Docker容器的管理请求。

Docker Client可以通过以下三种方式和Docker Daemon建立通信，分别为：tcp：//host：port、unix：//path_to_socket和fd：//socketfd。为简单起见，本书主要使用第一种方式作为讲述两者通信的原型。通信方式确定后，DockerClient与Docker Daemon建立连接并传输请求时，可以通过命令行flag参数的形式，设置安全传输层协议（TLS）的有关参数，保证传输的安全性。

Docker Client发送容器管理请求后，请求由Docker Daemon接收并处理，当Docker Client接收到返回的请求响应并做简单处理后，Docker Client一次完整的生命周期就此结束。若需要继续发送容器管理请求，用户必须再次通过可执行文件docker创建Docker Client，并走完以上相同的流程。

1.3.2　Docker Daemon

Docker Daemon是Docker架构中一个常驻在后台的系统进程。所谓的“运行Docker”，即代表运行Docker Daemon。总之，DockerDaemon的作用主要有以下两方面：

·接收并处理Docker Client发送的请求。

·管理所有的Docker容器。

Docker Daemon运行时，会在后台启动一个Server，Server负责接收Docker Client发送的请求；接收请求后，Server通过路由与分发调度，找到相应的Handler来处理请求。

启动Docker Daemon所使用的可执行文件同样是docker，与Docker Client启动所使用的可执行文件docker相同。既然Docker Client与Docker Daemon都可以通过docker二进制文件创建，那么如何辨别两者就变得非常重要。实际上，执行docker命令时，通过传入的参数可以辨别Docker Daemon与Docker Client，如docker–d代表Docker Daemon的启动，dockerps则代表创建Docker Client，并发送ps请求。

Docker Daemon的架构大致可以分为三部分：Docker Server、Engine和Job。Daemon的架构如图1-2所示。

1.Docker Server

Docker Server在Docker架构中专门服务于Docker Client，它的功能是接收并调度分发Docker Client发送的请求。Docker Server的架构如图1-3所示。

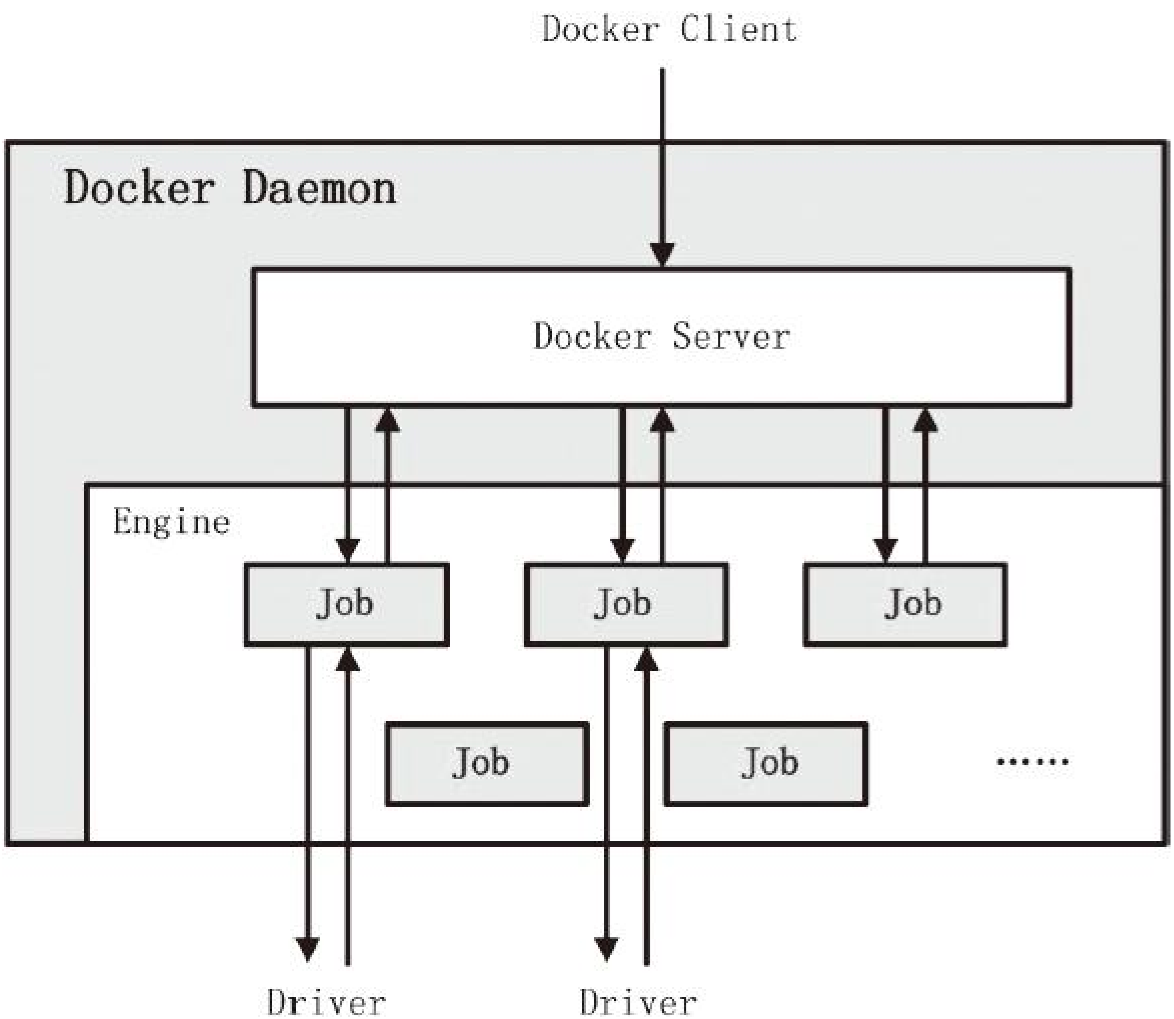


图1-2　Docker Daemon架构示意图

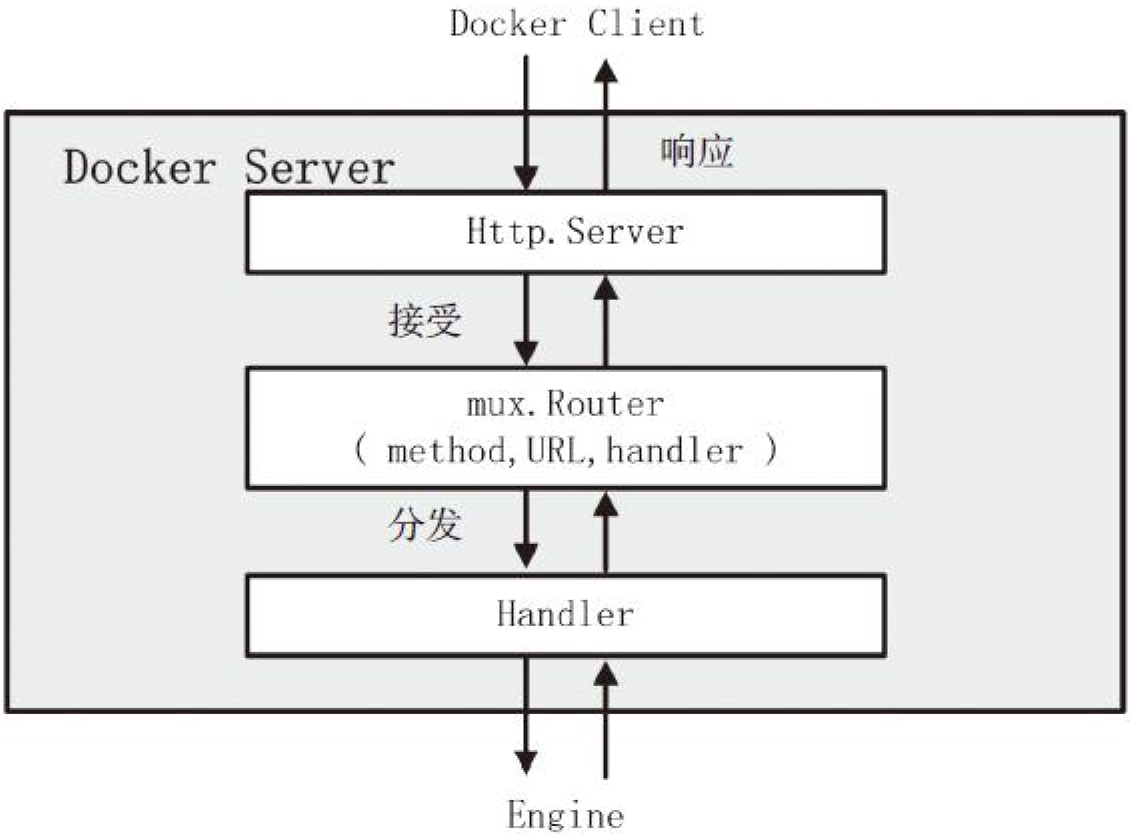


图1-3　Docker Server架构示意图

在Docker Daemon的启动过程中，DockerServer第一个完成。Docker Server通过包gorilla/mux，创建了一个mux.Router路由器，提供请求的路由功能。在Go语言中，gorilla/mux是一个强大的URL路由器以及调度分发器。创建路由器之后，Docker Server为mux.Router中添加有效的路由项，每一个路由项由HTTP请求方法（PUT、POST、GET或DELETE）、URL和Handler三部分组成。

由于Docker Client通过HTTP协议访问Docker Daemon，故DockerServer创建完mux.Router之后，将Server的监听地址以及

mux.Router作为参数，创建一个httpSrv= http.Server{}，最终执行httpSrv.Serve（ ） 开始服务于外部请求。

在服务过程中，Docker Server在listener上接收Docker Client的访问请求。对于每一个Docker Client请求，DockerServer均会创建一个全新的goroutine来服务。在goroutine中，Docker Server首先读取请求内容，然后做请求解析工作，接着匹配相应的路由项，随后调用相应的Handler来处理，最后Handler处理完请求之后给Docker Client回复响应。

需要注意的是：Docker Server在Docker的启动过程中运行，通过一个名为“serveapi”的Job来实现。理论上，Docker Server的运行只是众多Job中的一个，但是为了强调Docker Server的重要性以及它为后续Job服务的重要特性，本书将“serveapi”的Job单独抽离出来分析，理解为Docker Server。

2.Engine

Engine是Docker架构中的运行引擎，同时也是Docker运行的核心模块。Engine存储着大量的容器信息，同时管理着Docker大部分Job的执行。换言之，Docker中大部分任务的执行都需要Engine协助，并通过Engine匹配相应的Job完成Job的执行。

在Docker源码中，有关Engine的数据结构定义中含有一个名为handlers的对象。该handlers对象存储的是关于众多特定Job各自的处理方式handler。举例说明，Engine的handlers对象中有一项为：{"create"：daemon.ContainerCreate，}，则说明当执行名为“create”的Job时，执行的是daemon.ContainerCreate这个handler。

除了容器管理之外，Engine还接管Docker Daemon的某些特定任务。当Docker Daemon遭遇到自身进程需要退出的情况时，Engine还负责完成DockerDaemon退出前的所有善后工作。

3.Job

Job可以认为是Docker架构中Engine内部最基本的工作执行单元。DockerDaemon可以完成的每一项工作都会呈现为一个Job。例如，在Docker容器内部运行一个进程，这是一个Job；创建一个新的容器，这是一个Job；在网络上下载一个文档，这是一个Job；包括之前在Docker Server部分谈及的，创建Server服务于HTTP协议的API，这也是一个Job，等等。

有关Job接口的设计，与UNIX进程非常相仿。比如说，Job有一个名称，有运行时参数，有环境变量，有标准输入与标准输出，有标准错误，还有返回状态等。

对于Job而言，定义完毕之后，运行才能完成Job自身真正的使命。Job的运行函数Run（ ）则用以执行Job本身。

1.3.3 Docker Registry

Docker Registry是一个存储容器镜像(Docker Image) 的仓库。容器镜像(Docker Image) 是容器创建时用来初始化容器rootfs的文件系统内容。Docker Registry将大量的容器镜像汇集在一起，并为分散的Docker Daemon提供镜像服务。

Docker的运行过程中，有三种情况可能与Docker Registry通信，分别为搜索镜像、下载镜像、上传镜像。这三种情况所对应的Job名称分别为search、pull和push。

不同场景下，Docker Daemon可以使用不同的Docker Registry。公有Registry与私有Regsitry就是两种场景模式不同的Docker Registry。其中，大家熟知的Docker Hub，就是全球范围内最大的公有Registry。Docker可以通过互联网访问Docker Hub，并下载容器镜像；同时Docker也允许用户构建本地私有Registry，使容器镜像的获取在内网完成。

1.3.4 Graph

Graph在Docker架构中扮演的角色是容器镜像的保管者。不论是Docker下载的镜像，还是Docker构建的镜像，均由Graph统一化管理。由于Docker支持多种不同的镜像存储方式，如aufs、devicemapper、Btrfs等，故Graph对镜像的存储也会因以上种类而存在一些差异。对Docker而言，同一种类型的镜像被称为一个repository，如名称为ubuntu的镜像都同属一个repository；而同一个repository下的镜像则会因tag存在差异而不同，如ubuntu这个repository下有tag为12.04的镜像，也有tag为14.04的镜像。Docker中Graph的架构如图1-4所示。

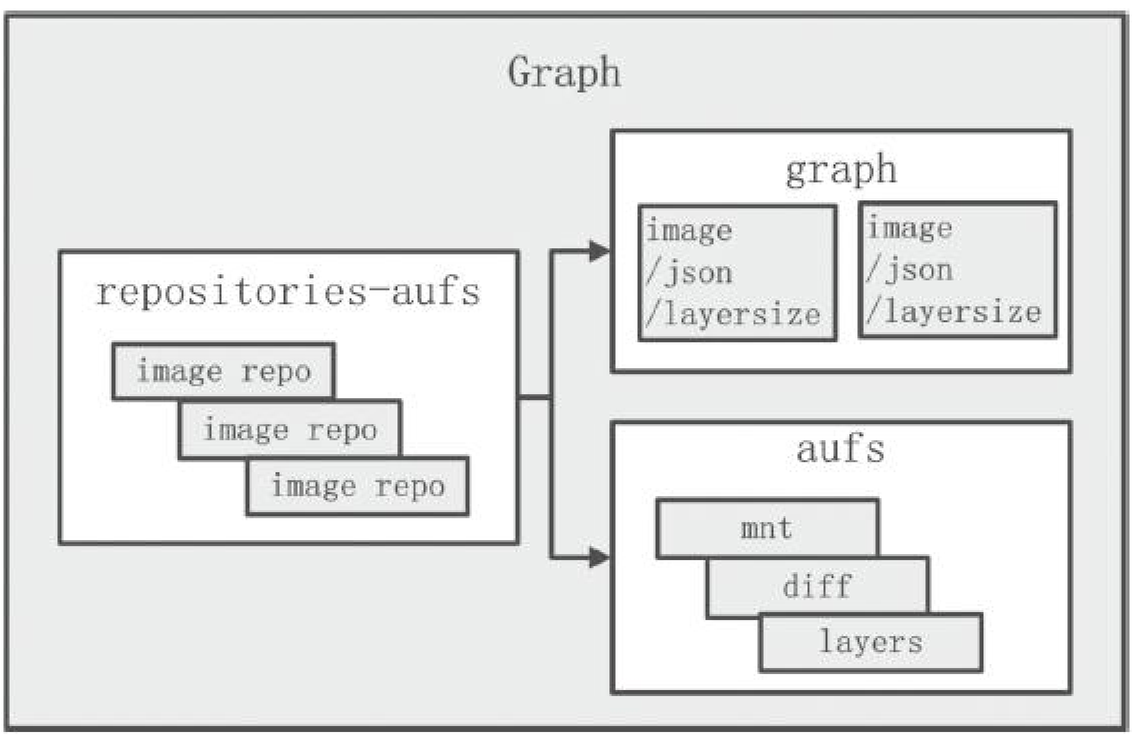


图1-4　Graph架构示意图

本书对Graph以及容器镜像（Docker Image）的分析将以aufs为主，并在第8章进行深入分析。

1.3.5 Driver

Driver是Docker架构中的驱动模块。通过Driver驱动，Docker可以实现对Docker容器运行环境的定制，定制的维度主要有网络环境、存储方式以及容器执行方式。需要注意的是，Docker运行的生命周期中，并非用户所有的操作都是针对Docker容器的管理，同时包括用户对Docker运行信息的获取，还包括Docker对Graph的存储与记录等。因此，为了将仅与Docker容器有关的管理从Docker Daemon的所有逻辑中区分开来，Docker的创造者设计了Driver层来抽象不同类别各自的功能范畴。

Docker Driver的实现可以分为以下三类驱动：graphdriver、networkdriver和execdriver。

graphdriver主要用于完成容器镜像的管理，包括从远程Docker Registry上下载镜像并进行存储，也包括本地构建完镜像后的存储。当用户下载指定的容器镜像时，graphdriver将容器镜像分层存储在本地的指定目录下；同时当用户需要使用指定的容器镜像来创建容器时，graphdriver从本地镜像存储目录中获取指定的容器镜像，并按特定规则为容器准备rootfs；另外，当用户需要通过指定Dockerfile构建全新镜像时，graphdriver会负责新镜像的存储管理。

在graphdriver的初始化过程之前，有4种文件系统或类文件系统的驱动Driver在DockerDaemon内部注册，它们分别是aufs、btrfs、vfs和devmapper。其中，aufs、btrfs以及devmapper用于容器镜像的管理，vfs用于容器volume的管理。Docker在初始化之时，优先通过获取系统环境变量“DOCKER_DRIVER”来提取所使用driver的指定类型。因此，之后所有的Graph操作，都使用该driver来执行。Docker镜像是Docker技术中非常关键的。2014年12月，在Linux 3.18-rc2版本中OverlayFS被合并至Linux内核主线，在Docker 1.4.0版本发布时，Docker官方宣布支持overlay这一类graphdriver，即用户在启动Docker Daemon时可以选择制定graphdriver的类型为overlay。graphdriver的架构如图1-5所示。

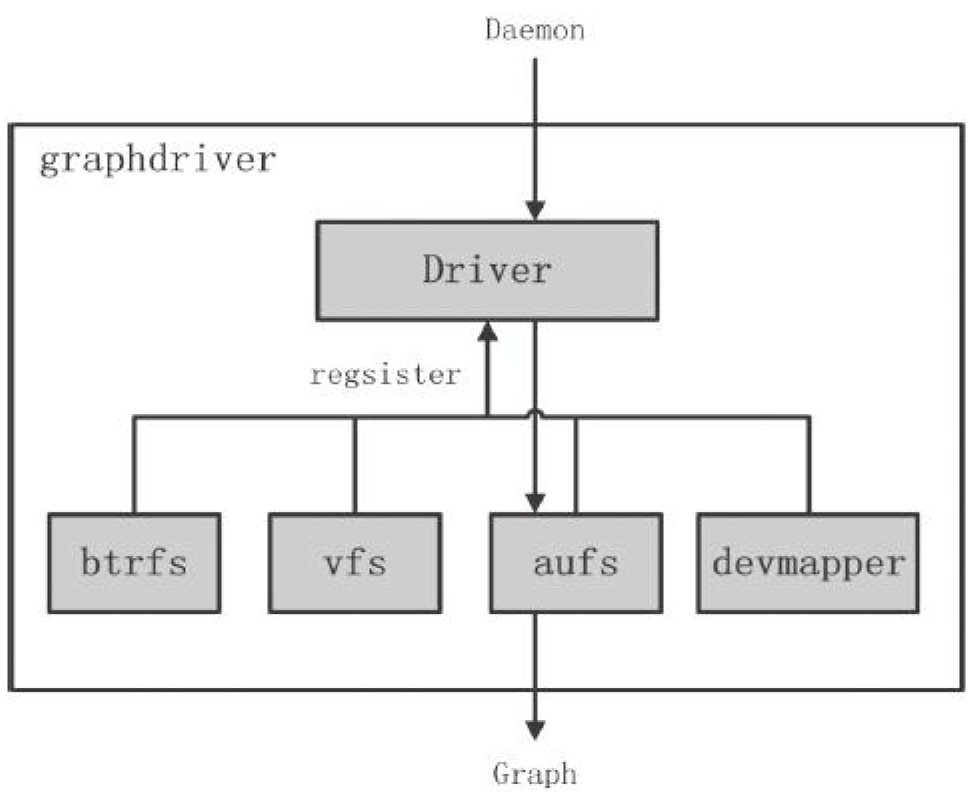


图1-5　graphdriver架构示意图

networkdriver的作用是完成Docker容器网络环境的配置，其中包括Docker Daemon启动时为Docker环境创建网桥；Docker容器创建前为其分配相应的网络接口资源；以及为Docker容器分配IP、端口并与宿主机做NAT端口映射，设置容器防火墙策略等。networkdriver的架构如图1-6所示。

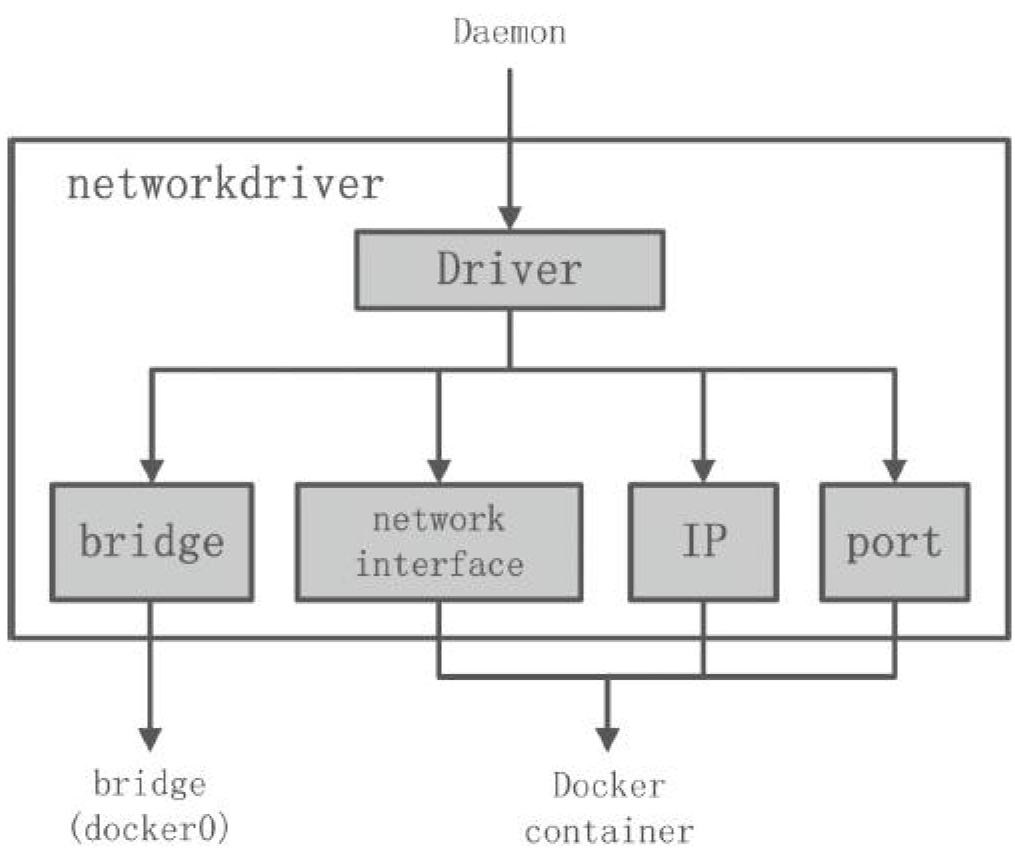


图1-6　networkdriver架构示意图

execdriver作为Docker容器的执行驱动，负责创建容器运行时的命名空间，负责容器资源使用的统计与限制，负责容器内部进程的

真正运行等。在Docker 0.9.0版本之前，execdriver只能通过LXC驱动来实现容器的启动管理。实际上，当时Docker通过LXC驱动调用Linux下的LXC工具管理容器的创建，并控制管理容器的生命周期。从Docker 0.9.0开始，在继续支持LXC的情况下，Docker的execdriver默认使用native驱动，native驱动完全独立于LXC，属于Docker项目下第一个全新的子项目，用于容器的创建与管理。Docker默认使用native驱动的具体体现为：Docker Daemon启动过程中加载的ExecDriverflag参数在配置文件中已经被设为native。native这个execdriver的存在，使得Docker对Linux容器的创建与管理有了自己的解决方案。execdriver架构如图1-7所示。

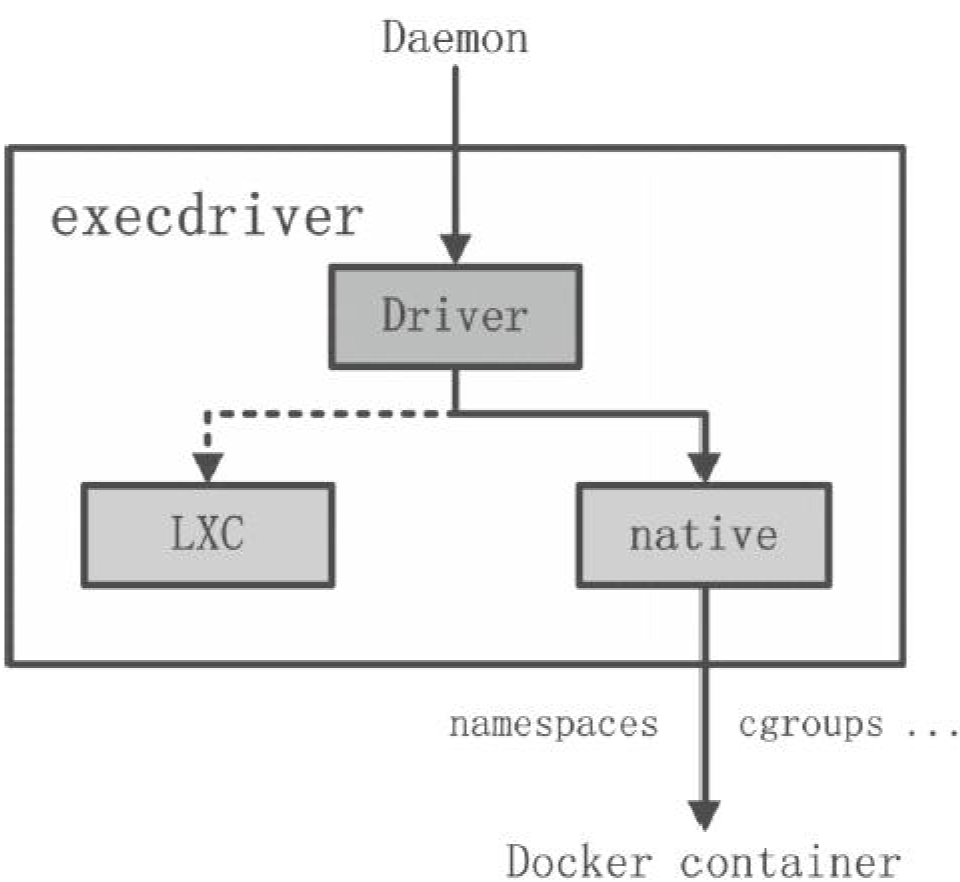


图1-7　execdriver架构示意图

1.3.6 libcontainer

libcontainer是Docker架构中一个使用Go语言设计实现的库，设计初衷是希望该库可以不依靠任何依赖，直接访问内核中与容器相关的系统调用。

正是由于libcontainer的存在，Docker可以直接调用libcontainer，而最终操作容器的namespaces、cgroups、apparmor、网络设备以及防火墙规则等。这一系列操作的完成都不需要依赖LXC或者其他包。libcontainer架构如图1-8所示。

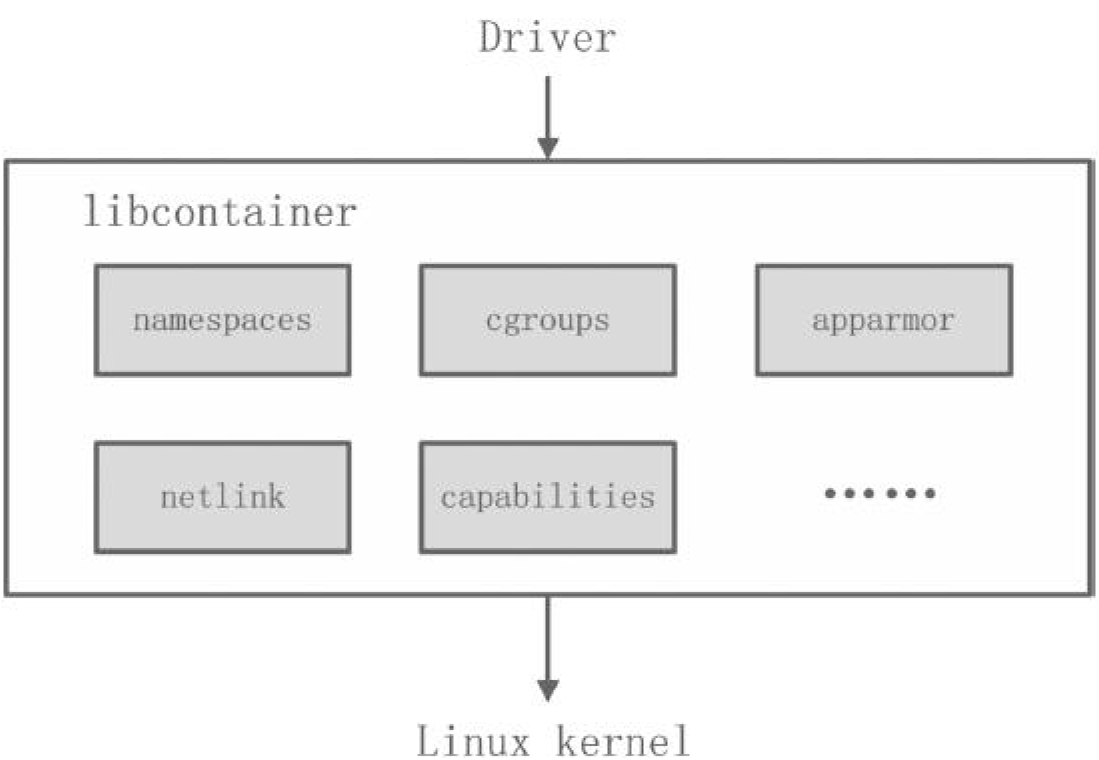


图1-8　libcontainer示意图

另外，libcontainer提供了一整套标准的接口来满足上层对容器管理的需求。或者说，libcontainer屏蔽了Docker上层对容器的直接管理。又由于libcontainer使用Go这种跨平台的语言开发实现，且本身又可以被上层多种不同的编程语言访问，因此，很难说未来的Docker一定会与Linux平台紧紧捆绑在一起。Docker Daemon的逻辑完全有可能位于其他非Linux操作系统的平台上，仅仅通过libcontainer的远程调用来实现对Docker容器的管理。另一方面，libcontainer与Docker Daemon的松耦合设计，似乎让用户感受到了除Linux Container之外其他的容器技术接入Docker Daemon的可能性。libcontainer承接Linux内核与Docker Daemon的同时，也让Docker的生态在跨平台方面充满生机。与此同时，Microsoft在其著名云计算平台Azure中，也添加了对Docker的支持，可见Docker的开放程度与业界的火热度。

暂不谈Docker，由于本身完善的功能以及与应用系统的松耦合特性，libcontainer很有可能会在众多其他以容器为原型的平台出现，同时也很有可能催生出云计算领域全新的项目。

1.3.7 Docker Container

Docker Container（Docker容器）是Docker架构中服务交付的最终体现形式。Docker通过DockerDaemon的管理，libcontainer的执行，最终创建Docker容器。Docker容器作为一个交付单位，功能类似于传统意义上的虚拟机（Virtual Machine），具备资源受限、环境与外界隔离的特点。然而，实现手段却与KVM、Xen等传统虚拟化技术大相径庭。

Docker容器的从无到有，涉及Docker利用到的很多技术。总而言之，用户可以根据自己的需求，通过Docker Client向Docker Daemon发送容器的创建与启动请求，请求中将携带容器的配置信息，从而达到定制相应Docker容器的目的。用户对Docker容器的配置有以下4个基本方面：

·通过指定容器镜像，使得Docker容器可以自定义rootfs等文件系统。

·通过指定物理资源的配额，如CPU、内存等，使得Docker容器使用受限的物理资源。

·通过配置容器网络及其安全策略，使得Docker容器拥有独立且安全的网络环境。

·通过指定容器的运行命令，使得Docker容器执行指定的任务。

Docker容器示意图如图1-9所示。

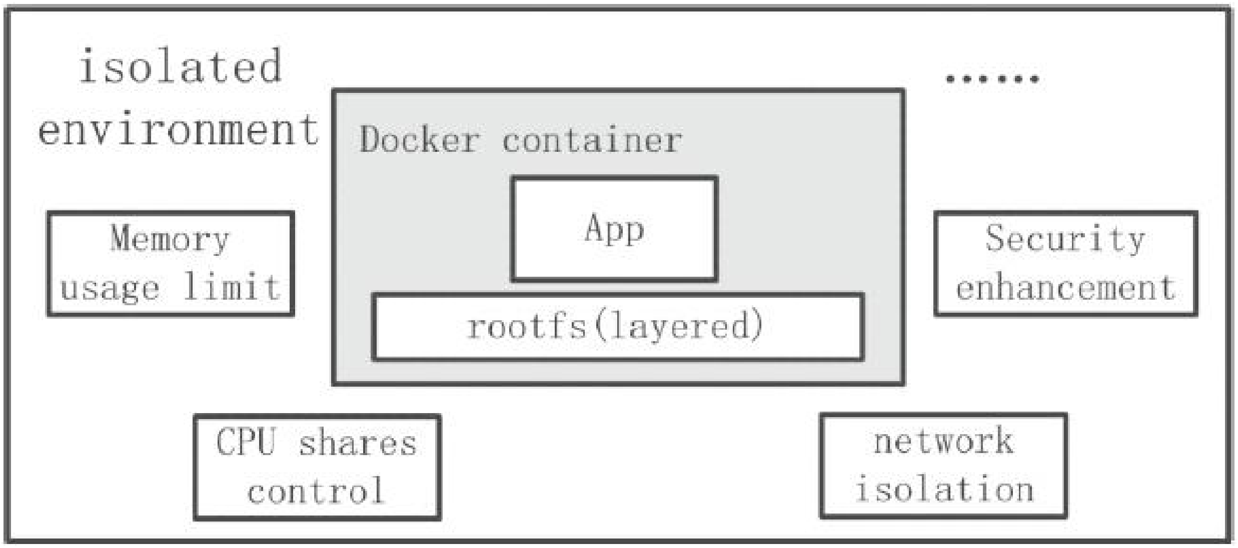


图1-9　Docker容器示意图

1.4　Docker运行案例分析

1.3节着重介绍了Docker架构中各个模块的功能，学完后我们可以对Docker的架构有一个宏观的认识。熟悉一款软件，研究一个系统，从静态的角度认识架构的各个模块，仅仅是第一步；从动态的角度，掌握软件或者系统的运行原理，即熟知架构中模块间的通信逻辑，无疑会让自己对软件或系统的理解更上一层楼。本节将从实际的Docker运行案例出发，串联Docker各模块，从而学习Docker的运行流程。分析原型为Docker中的docker pull与docker run两个命令。

1.4.1　docker pull

1.3节中我们提到，用户可以为容器指定镜像，作为容器运行时的rootfs，既然如此，镜像从何而来则成为一个关键。答案很简单，一切都归功于docker pull命令。

docker pull命令的作用是：Docker Daemon从Docker Registry下载指定的容器镜像，并将镜像存储在本地的Graph中，以备后续创建Docker容器时使用。docker pull命令的执行流程如图1-10所示。

图1-10中有编号的箭头表示docker pull命令在发起后，Docker架构中相应模块所做的一系列运行操作。下面我们逐一分析这些步骤。

1）Docker Client处理用户发起的docker pull命令，解析完请求以及参数之后，发送一个HTTP请求给Docker Server，HTTP请求方法为POST，请求URL为"/images/create？"+"xxx"，实际意义为下载相应的镜像。

2）Docker Server接收以上HTTP请求，并交给mux.Router，mux.Router通过URL以及请求方法类型来确定执行该请求的具体

handler。

3） mux.Router将请求路由分发至相应的handler，具体为PostImagesCreate。

4） 在PostImageCreate这个handler之中，创建并初始化一个名为"pull"的Job，之后触发执行该Job。

5） 名为"pull"的Job在执行过程中执行pullRepository操作，即从Docker Registry中下载相应的一个或者多个Docker镜像。

6） 名为"pull"的Job将下载的Docker镜像交给graphdriver管理。

7） graphdriver负责存储Docker镜像，一方面将实际镜像存储至本地文件系统中，另一方面为镜像创建对象，由Docker Daemon统一管理。

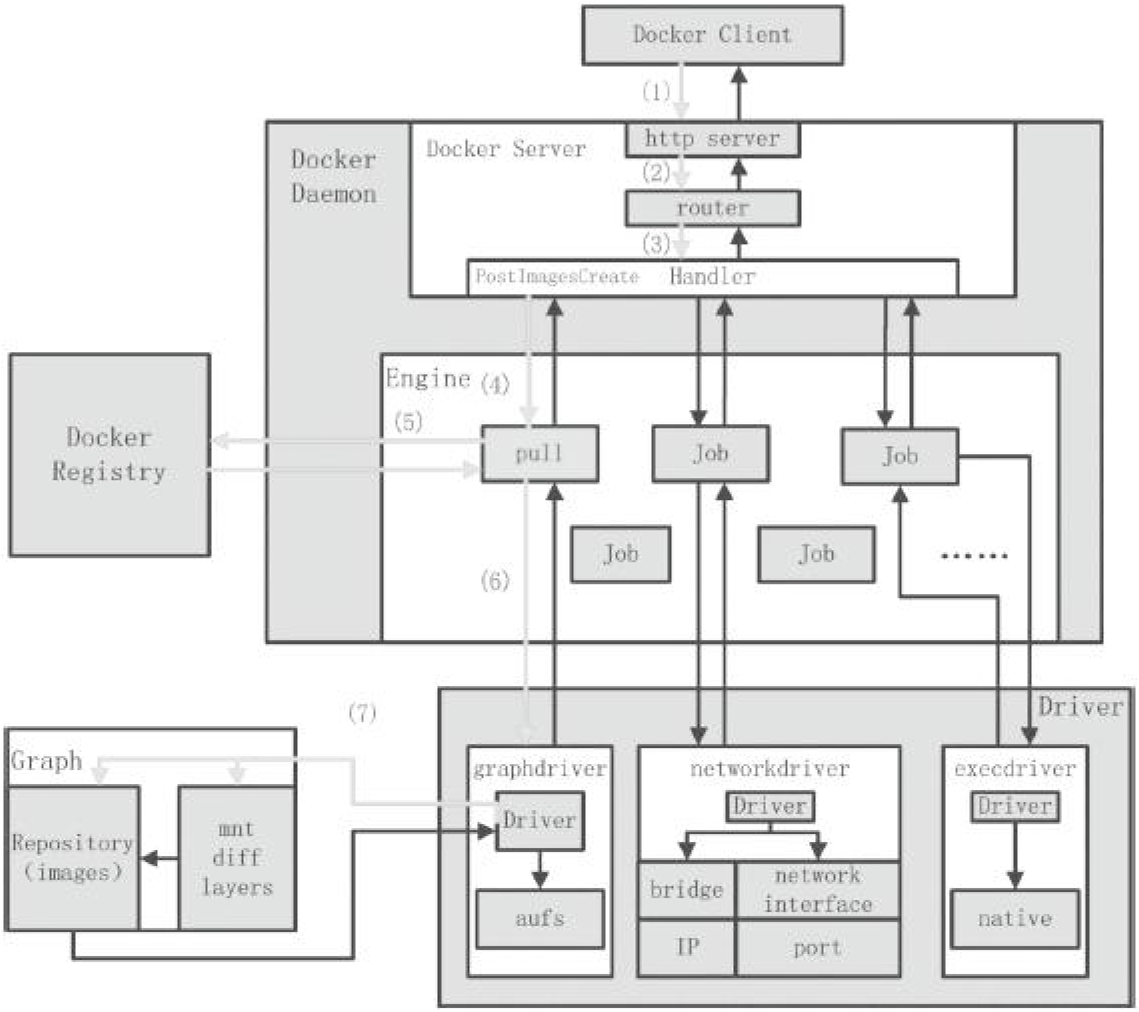


图1-10　docker pull命令执行流程示意图

1.4.2　docker run

docker run命令的作用是创建一个全新的Docker容器，并在容器内部运行指定命令。Docker Daemon处理用户发起的这条命令时，所做工作可以分为两部分：第一，创建Docker容器对象，并为容器准备所需的rootfs；第二，创建容器的运行环境，如网络环境、资源限制等，最终真正运行用户指令。因此，在dockerrun命令的完整执行流程中，Docker Client给Docker Server发送了两次HTTP请求，第二次请求的发起取决于第一次请求的返回状态。docker run命令执行流程如图1-11所示。

架构中相应模块所做的一系列运行。

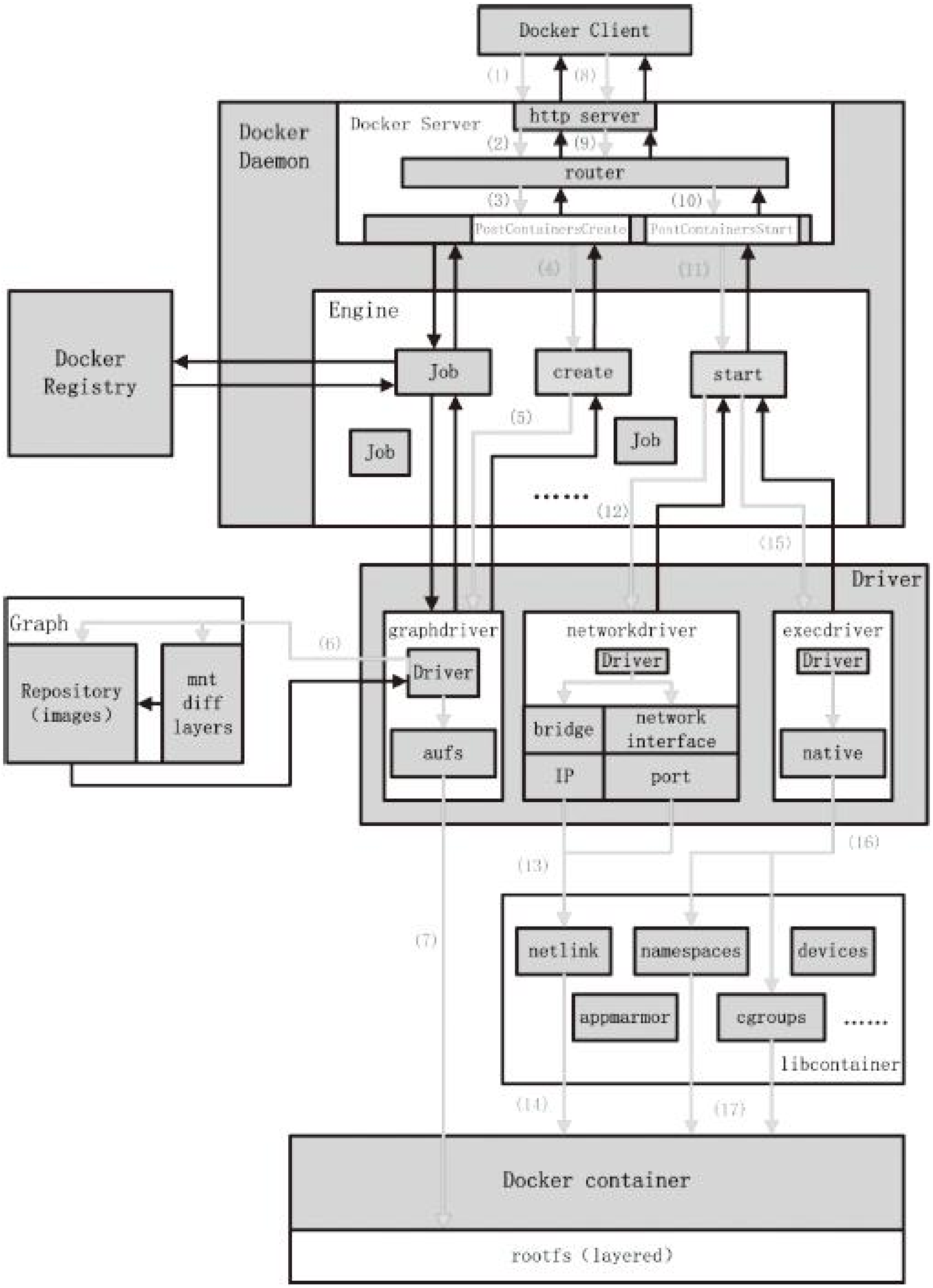


图1-11　docker run命令执行流程示意图

图1-11中有编号的箭头表示dockerrun命令在发起后，Docker架构中相应模块所做的一系列运行。下面我们逐一分析这些步骤：

1） Docker Client处理用户发起的docker run命令，解析完请求与参数之后，向Docker Server发送一个HTTP请求，HTTP请求方法为POST，请求URL为"/containers/create？"+"xxx"，实际意义为创建一个容器对象，即Docker Daemon程序逻辑中的容器对象，并非实际运行的容器。

2） Docker Server接收以上HTTP请求，并交给mux.Router，mux.Router通过URL以及请求方法来确定执行该请求的具体handler。

3） mux.Router将请求路由分发至相应的handler，具体为PostContainersCreate。

4） 在PostContainersCreate这个handler之中，创建并初始化一个名为"create"的Job，之后触发执行该Job。

5） 名为"create"的Job在运行过程中执行Container.Create操作，该操作需要获取容器镜像来为Docker容器准备rootfs，通过graphdriver完成。

6） graphdriver从Graph中获取创建Docker容器rootfs所需要的所有镜像。

7） graphdriver将rootfs的所有镜像通过某种联合文件系统的方式加载至Docker容器指定的文件目录下。

8） 若以上操作全部正常执行，没有返回错误或异常，则Docker Client收到Docker Server返回状态之后，发起第二次HTTP请求。请求方法为"POST"，请求URL为"/containers/"+container_ID+"/start"，实际意义为启动时才创建完毕的容器对象，实现物理容器的真正运行。

9） Docker Server接收以上HTTP请求，并交给mux.Router，mux.Router通过URL以及请求方法来确定执行该请求的具体handler。

10） mux.Router将请求路由分发至相应的handler，具体为PostContainersStart。

11） 在PostContainersStart这个handler之中，创建并初始化名为"start"的Job，之后触发执行该Job。

12） 名为"start"的Job执行需要完成一系列与Docker容器相关的配置工作，其中之一是为Docker容器网络环境分配网络资源，如IP资源等，通过调用networkdriver完成。

13） networkdriver为指定的Docker容器分配网络资源，其中有IP、port等，另外为容器设置防火墙规则。

14） 返回名为"start"的Job，执行完一些辅助性操作后，Job开始执行用户指令，调用execdriver。

15） execdriver被调用，开始初始化Docker容器内部的运行环境，如命名空间、资源控制与隔离，以及用户命令的执行，相应的操作转交至libcontainer来完成。

16） libcontainer被调用，完成Docker容器内部的运行环境初始化，并最终执行用户要求启动的命令。

1.5 总结

本章从Docker 1.2.0的源码入手，首先分析抽象出Docker的架构图，并对该架构图中的各个模块进行功能与实现的简要分析，最后通过Docker的两个基础命令展示了Docker内部的运行。

通过对Docker架构的学习，可以全面深化对Docker设计、功能与价值的理解。同时在借助Docker实现用户定制的分布式系统时，也能更好地找到已有平台与Docker较为理想的契合点。另外，熟悉Docker现有架构以及设计思想，也能为云计算PaaS领域带来更多的启示，催生出更多创新想法。

第2章　Docker Client创建与命令执行

2.1　引言

如今，作为业界领先的轻量级虚拟化容器管理引擎，Docker给全球开发者提供了一种新颖、便捷的软件集成测试与部署之道。团队开发软件时，Docker可以提供可复用的运行环境、灵活的资源配置、便捷的集成测试方法，以及一键式的部署方式。可以说，Docker在简化持续集成、运维部署方面将其功能发挥得淋漓尽致，它让开发者从重复的持续集成、运维部署中完全解放出来，把精力真正地倾注在开发上。

然而，要把Docker的功能发挥到极致，并非一件易事。在深刻理解Docker架构的情况下，熟练掌握Docker Client的使用也非常有必要。前者可以参阅第1章，本章主要针对后者，从源码的角度分析Docker Client，力求帮助开发者更深刻地理解Docker Client的具体实现，最终更好地掌握Docker Client的使用方法。

本章基于Docker 1.2.0的源码，分析Docker Client的内容。主要包括两个部分，分别是DockerClient的创建与Docker Client对命令的执行。两部分分析的具体内容如下。

第一部分分析Docker Client的创建。这部分的分析可分为以下三个步骤：

·分析如何通过docker命令，解析出命令行flag参数，以及docker命令中的请求参数。

·分析如何处理具体的flag参数信息，并收集Docker Client所需的配置信息。

·分析如何创建一个Docker Client。

第二部分在已有Docker Client的基础上，分析如何执行docker命令。这部分的分析又可分为以下两个步骤。

·分析如何解析docker命令中的请求参数，获取相应请求的类型。

·分析Docker Client如何执行具体的请求命令，最终将请求发送至Docker Server。

2.2 创建Docker Client

对于Docker这样一个Client/Server的架构，客户端的存在意味着Docker相应任务的发起。用户首先需要创建一个DockerClient，随后将特定的请求类型与参数传递至Docker Client，最终由Docker Client转义成Docker Server能识别的形式，并发送至Docker Server。

Docker Client的创建实质上是Docker用户通过二进制可执行文件docker，创建与Docker Server建立联系的客户端。以下分3个小节分别阐述Docker Client的创建流程。

Docker Client完整的运行流程如图2-1所示。

执行请求的过程。

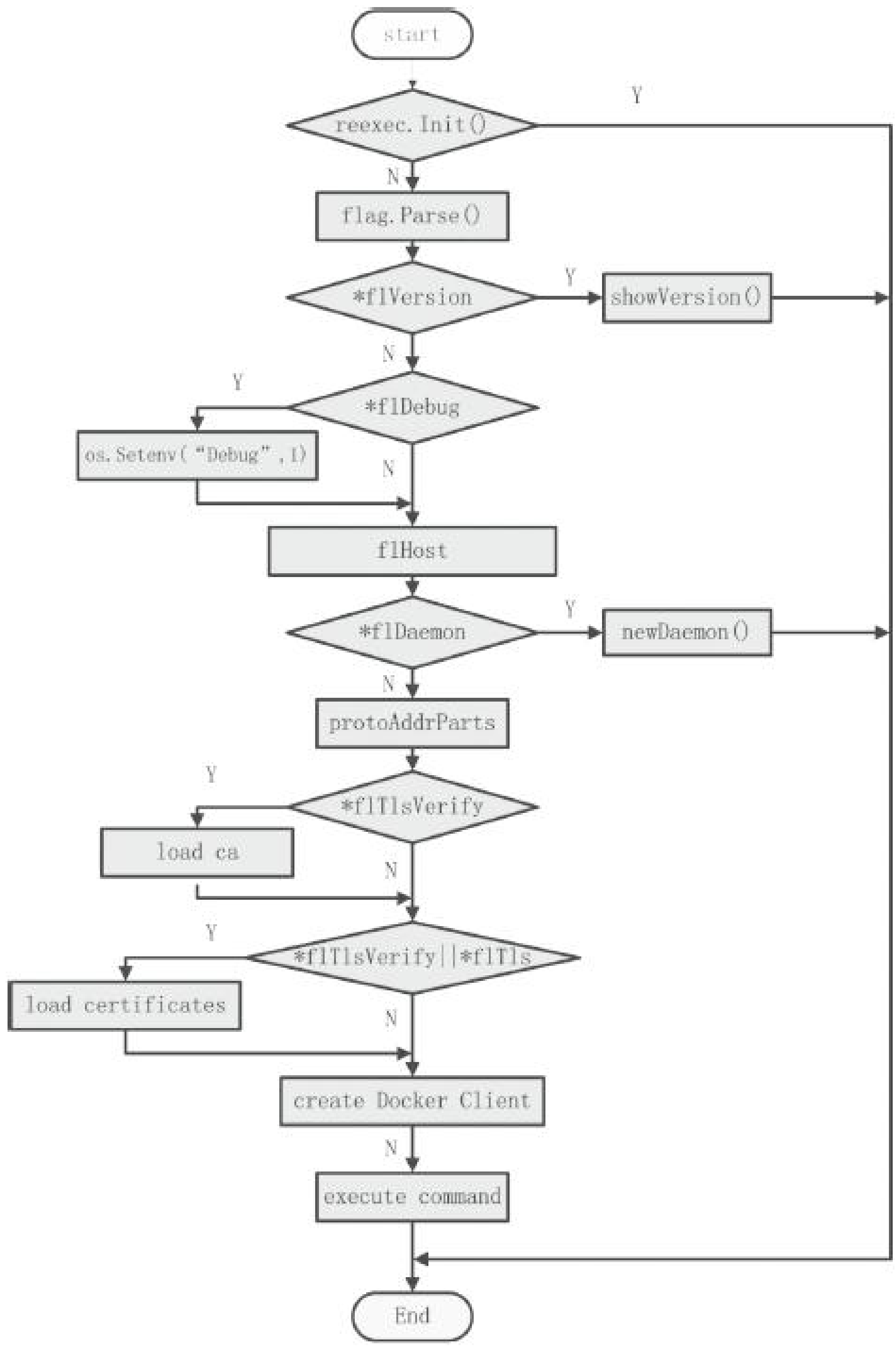


图2-1　DockerClient的运行流程

通过学习图2-1，我们可以更为清晰地了解Docker Client创建及执行请求的过程。其中涉及诸多Docker源码层次中的专有名词，本章

后续会一一解释与分析。

2.2.1　Docker命令的flag参数解析

众所周知，在Docker的具体实现中，Docker Server与Docker Client均由可执行文件docker来完成创建并启动。那么，了解docker可执行文件通过何种方式来区分到底是Docker Server还是Docker Client，就显得尤为重要。

首先通过docker命令举例说明其中的区别。Docker Server的启动，命令为docker-d或docker--daemon=true；而Docker Client的启动则体现为docker--daemon=false ps、docker pull NAME等。

其实，对于Docker请求中的参数，我们可以将其分为两类：第一类为命令行参数，即docker程序运行时所需提供的参数，如：-D、--daemon=true、--daemon=false等；第二类为docker发送给Docker Server的实际请求参数，如：ps、pull NAME等。

对于第一类，我们习惯将其称为flag参数，在Go语言的标准库中，专门为该类参数提供了一个flag包，方便进行命令行参数的解析。

清楚docker二进制文件的使用以及基本的命令行flag参数之后，我们可以进入实现Docker Client创建的源码中，位于./docker/docker/docker.go。这个go文件包含了整个Docker的main函数，也就是整个Docker（不论Docker Daemon还是Docker Client）的运行入口。部分main函数代码如下：

```
func main() {
     if reexec.Init() {
          return
     }
     flag.Parse()
     // FIXME: validate daemon flags here
     ...
}
```

以上源码实现中，首先判断reexec.Init（）方法的返回值，若为真，则直接退出运行，否则将继续执行。reexec.Init（）函数的定义位于./docker/reexec/reexec.go，可以发现由于在docker运行之前没有任何Initializer注册，故该代码段执行的返回值为假。reexec存在的作用是：协调execdriver与容器创建时dockerinit这两者的关系。第13章在分析dockerinit的启动时，将详细讲解reexec的作用。

判断reexec.Init（）之后，Docker的main函数通过调用flag.Parse（）函数，解析命令行中的flag参数。如果熟悉Go语言中的flag参数，一定知道解析flag参数的值之前，程序必须先定义相应的flag参数。进一步查看Docker的源码，我们可以发现Docker

在./docker/docker/flag.go中定义了多个flag参数，并通过init函数进行部分flag参数的初始化。代码如下：

```
var (
         flVersion = flag.Bool([]string{"v", "-version"}, false, "Print version
information and quit")
    flDaemon = flag.Bool([]string{"d", "-daemon"}, false, "Enable daemon mode")
    flDebug = flag.Bool([]string{"D", "-debug"}, false, "Enable debug mode")
         flSocketGroup = flag.String([]string{"G", "-group"}, "docker", "Group to
assign the unix socket specified by -H when running in daemon modeuse '' (the
empty string) to disable setting of a group")
         flEnableCors = flag.Bool([]string{"#api-enable-cors", "-api-enable-
cors"}, false, "Enable CORS headers in the remote API")
         flTls = flag.Bool([]string{"-tls"}, false, "Use TLS; implied by tls-
verify flags")
         flTlsVerify = flag.Bool([]string{"-tlsverify"}, false, "Use TLS and
verify the remote (daemon: verify client, client: verify daemon)")
         // these are initialized in init() below since their default values
depend on dockerCertPath which isn't fully initialized until init() runs
    flCa    *string
    flCert  *string
    flKey   *string
    flHosts []string
    )
    func init() {
         flCa = flag.String([]string{"-tlscacert"}, filepath.Join(dockerCertPath,
defaultCaFile), "Trust only remotes providing a certificate signed by the CA
given here")
         flCert = flag.String([]string{"-tlscert"}, filepath.Join(dockerCertPath,
defaultCertFile), "Path to TLS certificate file")
         flKey = flag.String([]string{"-tlskey"}, filepath.Join(dockerCertPath,
defaultKeyFile), "Path to TLS key file")
         opts.HostListVar(&flHosts, []string{"H", "-host"}, "The socket(s) to
bind to in daemon mode\nspecified using one or more tcp://host:port,
unix:///path/to/socket, fd://* or fd://socketfd.")
}
```

以上源码展示了Docker如何定义flag参数，以及在init函数中实现部分flag参数的初始化。Docker的main函数执行前，这些变量创建以及初始化工作已经全部完成。这里涉及了Go语言的一个特性，即init函数的执行。Go语言中引入其他包（import package）、变量的定义、init函数以及main函数这四者的执行顺序如图2-2所示。

关于Golang中的init函数，深入分析可以得出以下特性：

·init函数用于程序执行前包的初始化工作，比如初始化变量等；

·每个包可以有多个init函数；

·包的每一个源文件也可以有多个init函数；

·同一个包内的init函数的执行顺序没有明确的定义；

·不同包的init函数按照包导入的依赖关系决定初始化的顺序；

·init函数不能被调用，而是在main函数调用前自动被调用。

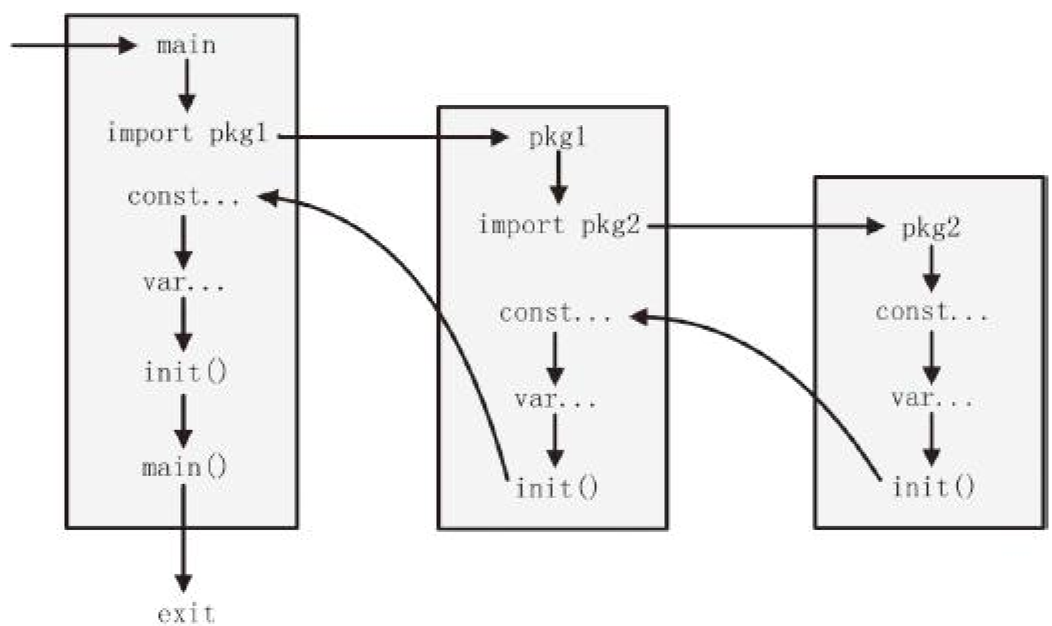


图2-2　Go语言程序加载顺序图

清楚Go语言一些基本的特性之后，回到Docker中来。Docker的main函数执行之前，Docker已经定义了诸多flag参数，并对很多flag参数进行初始化。定义并初始化的命令行flag参数有：flVersion、flDaemon、flDebug、flSocketGroup、flEnableCors、flTls、flTlsVerify、flCa、flCert、flKey、flHosts等。

以下具体分析flDaemon：

·定义：flDaemon＝flag.Bool（[]string{"d"，"-daemon"}，false，"Enable daemon mode"）；

·flDaemon的类型为Bool类型；

·flDaemon名称为"d"或者"-daemon"，该名称会出现在docker命令中，如docker–d；

·flDaemon的默认值为false；

·flDaemon的用途信息为"Enable daemon mode"；

·访问flDaemon的值时，使用指针*flDaemon解引用访问。

在解析命令行flag参数时，以下语句为合法的（以flDaemon为例）：

·-d，--daemon

·-d=true，--daemon=true

·-d="true"，--daemon="true"

·-d='true'，--daemon='true'

当解析到第一个非定义的flag参数时，命令行flag参数解析工作结束。举例说明，当执行docker命令docker--daemon=false--version=false ps时，flag参数解析主要完成两个工作：

·完成命令行flag参数的解析，根据flag的名称-daemon和-version，得知具体的flag参数为flDaemon和flVersion，并获得相应的值，均为false。

·遇到第一个非定义的flag参数ps时，flag包会将ps及其之后所有的参数存入flag.Args（），以便之后执行Docker Client具体的请求时使用。

如需深入学习flag的实现，可以参见Docker源码./docker/pkg/mflag/flag.go。

2.2.2 处理flag信息并收集Docker Client的配置信息

理解Go语言解析flag参数的相关知识，可以很大程度上帮助理解Docker的main函数的执行流程。通过总结，首先列出源码中处理的flag信息以及收集Docker Client的配置信息，然后再一一进行分析：

·处理的flag参数有：flVersion、flDebug、flDaemon、flTlsVerify以及flTls。

·为Docker Client收集的配置信息有：protoAddrParts（通过flHosts参数获得，作用是提供Docker Client与Docker Server的通信协议以及通信地址）、tlsConfig（通过一系列flag参数获得，如*flTls、*flTlsVerify，作用是提供安全传输层协议的保障）。

清楚flag参数以及Docker Client的配置信息之后，我们进入main函数的源码，具体分析如下。

在flag.Parse（）之后的源码如下：

```
if *flVersion {
     showVersion()
     return
}
```

以上代码很好理解，解析flag参数后，若Docker发现flag参数flVersion为真，则说明Docker用户希望查看Docker的版本信息。此时，Docker调用showVersion（ ）显示版本信息，并从main函数退出；否则的话，继续往下执行。

```
if *flDebug {
     os.Setenv("DEBUG", "1")
}
```

若flDebug参数为真的话，Docker通过os包中的Setenv函数创建一个名为DEBUG的环境变量，并将其值设为"1"；继续往下执行。

```
if len(flHosts) == 0 {
      defaultHost := os.Getenv("DOCKER_HOST")
      if defaultHost == "" || *flDaemon {
           // If we do not have a host, default to unix socket
           defaultHost = fmt.Sprintf("unix://%s", api.DEFAULTUNIXSOCKET)
      }
      if _, err := api.ValidateHost(defaultHost); err != nil {
           log.Fatal(err)
      }
      flHosts = append(flHosts, defaultHost)
}
```

以上的源码主要分析内部变量flHosts。flHosts的作用是为Docker Client提供所要连接的host对象，也就是为Docker Server提供所要监听的对象。

在分析过程中，首先判断flHosts变量是否长度为0。若是的话，则说明用户并没有显性传入地址，此时Docker的策略为选用默认值。Docker通过os包获取名为DOCKER_HOST环境变量的值，将其赋值

于defaultHost。若defaultHost为空或者flDaemon为真，说明目 前还没有一个定义的host对象，则将其默认设置为unix socket，值为api.DEFAULTUNIXSOCKET，该常量位于./docker/api/common.go，值为"/var/run/docker.sock"，故defaultHost为"unix：///var/run/docker.sock"。验证该defaultHost的合法性之后，将defaultHost的值追加至flHost的末尾，继续往下执行。当 然若flHost的长度不为0，则说明用户已经指定地址，同样继续往下执行。

```
if *flDaemon {
     mainDaemon()
     return
}
```

若flDaemon参数为真，则说明用户的需求是启动Docker Daemon。Docker随即执行mainDaemon函数，实现Docker Daemon的启动；若mainDaemon函数执行完毕，则退出main函数。一般mainDaemon函数不会主动终结，Docker Daemon将作为一个常驻进程运行在宿主机上。本章着重介绍Docker Client的启动，故假设flDaemon参数为假，不执行以上代码块。继续往下执行。

```
if len(flHosts) > 1 {
      log.Fatal("Please specify only one -H")
}
protoAddrParts := strings.SplitN(flHosts[0], "://", 2)
```

由于不执行Docker Daemon的启动流程，故属于Docker Client的执行逻辑。首先，判断flHosts的长度是否大于1。若flHosts的长度大于1，则说明需要新创建的Docker Client访问不止1个Docker Daemon地址，显然逻辑上行不通，故抛出错误日志，提醒用户只能指定一个Docker Daemon地址。接着，Docker将flHosts这个string数组中的第一个元素进行分割，通过"：//"来分割，分割出的两个部分放入变量protoAddrParts数组中。protoAddrParts的作用是：解析出Docker Client与Docker Server建立通信的协议与地址，为Docker Client创建过程中不可或缺的配置信息之一。一般情况下，flHosts[0]的值可以是tcp：//0.0.0.0：2375或者unix：///var/run/docker.sock等。

```
var (
     cli       *client.DockerCli
     tlsConfig tls.Config
)
tlsConfig.InsecureSkipVerify = true
```

由于之前已经假设过flDaemon为假，可以认定main函数的运行是为了Docker Client的创建与执行。Docker在这里创建了两个变量：一个为类型是*client.DockerCli的对象cli，另一个为类型是tls.Config的对象tlsConfig。定义完变量之后，Docker将tlsConfig的InsecureSkipVerify属性置为真。tlsConfig对象的创建是为了保障cli在传输数据的时候遵循安全传输层协议（TLS）。安全传输层协

议(TLS) 用于确保两个通信应用程序之间的保密性与数据完整性。tlsConfig是Docker Client创建过程中可选的配置信息。

```
// If we should verify the server, we need to load a trusted ca
if *flTlsVerify {
     *flTls = true
     certPool := x509.NewCertPool()
     file, err := ioutil.ReadFile(*flCa)
     if err != nil {
          log.Fatalf("Couldn't read ca cert %s: %s", *flCa, err)
     }
     certPool.AppendCertsFromPEM(file)
     tlsConfig.RootCAs = certPool
     tlsConfig.InsecureSkipVerify = false
}
```

若flTlsVerify这个flag参数为真，则说明Docker Client需Docker Server一起验证连接的安全性。此时，tlsConfig对象需要加载一个受信的ca文件。该ca文件的路径为*flCA参数的值，最终完成tlsConfig对象中RootCAs属性的赋值，并将InsecureSkipVerify属性置为假。

```
// If tls is enabled, try to load and send client certificates
if *flTls || *flTlsVerify {
     _, errCert := os.Stat(*flCert)
     _, errKey := os.Stat(*flKey)
     if errCert == nil && errKey == nil {
          *flTls = true
          cert, err := tls.LoadX509KeyPair(*flCert, *flKey)
          if err != nil {
               log.Fatalf("Couldn't load X509 key pair: %s. Key encrypted?", err)
          }
          tlsConfig.Certificates = []tls.Certificate{cert}
     }
}
```

如果flTls和flTlsVerify两个flag参数中有一个为真，则说明需要加载并发送客户端的证书。最终将证书内容交给tlsConfig的

Certificates属性。

至此，flag参数已经全部处理完毕，DockerClient也已经收集到所需的配置信息。下一节将主要分析如何创建Docker Client。

2.2.3 如何创建Docker Client

Docker Client的创建其实就是在已有配置参数信息的情况下，通过Client包中的NewDockerCli方法创建一个Docker Clinet实例cli。具体源码实现如下：

```
if *flTls ||  *flTlsVerify {


          cli = client.NewDockerCli(os.Stdin, os.Stdout, os.Stderr,
protoAddrParts[0], protoAddrParts[1], &tlsConfig) } else {


          cli = client.NewDockerCli(os.Stdin, os.Stdout, os.Stderr,
protoAddrParts[0], protoAddrParts[1], nil) }
```

若flag参数flTls为真或者flTlsVerify为真，则说明需要使用TLS协议来保障传输的安全性，故创建Docker Client的时候，将tlsConfig参数传入；否则，同样创建Docker Client，只不过tlsConfig为nil。

关于Client包中的NewDockerCli函数的实现，可以具体参见./docker/api/client/cli.go。

```
func NewDockerCli(in io.ReadCloser, out, err io.Writer, proto, addr string,
tlsConfig *tls.Config) *DockerCli {


var (
```

```
     isTerminal = false terminalFd uintptr scheme     = "http"

)

if tlsConfig != nil {

     scheme = "https"

}

if in != nil {

     if file, ok := out.(*os.File); ok {

          terminalFd = file.Fd() isTerminal = term.IsTerminal(terminalFd) }

}

if err == nil {

     err = out

}

return &DockerCli{

     proto:      proto, addr:       addr, in:         in, out:        out, err:
err, isTerminal: isTerminal, terminalFd: terminalFd, tlsConfig:  tlsConfig,
scheme:     scheme, }

}
```

总体而言，创建DockerCli对象的过程比较简单。较为重要的DockerCli的属性有：proto，DockerClient与Docker Server的传输协议；addr，Docker Client需要访问的host目 标地址；tlsConfig，安全传输层协议的配置。若tlsConfig不为空，则说明需要使用安全传输层协议，DockerCli对象的scheme设置为“https”，另外还有关于输入、输出以及错误显示的配置等。最终函数返回DockerCli对象。

通过调用client包中的NewDockerCli函数，程序最终创建了Docker Client，返回main包中的main函数之后，程序继续往下执行。

2.3　Docker命令执行

main函数执行到这个阶段，有以下内容需要为Docker命令的执行服务：创建完毕的Docker Client，docker命令中的请求参数（经flag解析后存放于flag.Arg（））。也就是说，程序需要使用Docker Client来分析Docker命令中的请求参数，得出请求的类型，转义为Docker Server可以识别的请求之后，最终发送给Docker Server。

Docker Client主要完成两方面的工作：解析请求命令，得出请求类型；执行具体类型的请求。本节将从这两个方面深入分析Docker Client。

2.3.1　Docker Client解析请求命令

Docker Client解析请求命令的工作，在Docker命令执行部分第一个完成。创建Docker Client之后，回到main函数中，继续执行的源码如下（位于./docker/docker/docker.go#L102-L110）：

```
if err := cli.Cmd(flag.Args()...); err != nil {
     if sterr, ok := err.(*utils.StatusError); ok {
          if sterr.Status != "" {
               log.Println(sterr.Status)
          }
          os.Exit(sterr.StatusCode)
     }
     log.Fatal(err)
}
```

学习以上源码可以发现，正如之前所说，Docker Client首先解析存放于flag.Args（）中的具体请求参数，执行的函数为cli对象的Cmd函数。Cmd函数的定义如下（位于./docker/api/client/cli.go#L51-L61）：

```
// Cmd executes the specified command
func (cli *DockerCli) Cmd(args ...string) error {
     if len(args) > 0 {
          method, exists := cli.getMethod(args[0])
          if !exists {
               fmt.Println("Error: Command not found:", args[0])
               return cli.CmdHelp(args[1:]...)
          }
          return method(args[1:]...)
     }
     return cli.CmdHelp(args...)
}
```

由代码注释可知，Cmd函数执行具体的指令。在源码实现中，首先判断请求参数列表的长度是否大于0。若长度不大于0，则说明没有请求信息，返回docker命令的Help信息；若长度大于1，则说明有请求信息，那么Docker Client首先通过请求参数列表中的第一个元素args[0]来获取具体的请求方法method。若上述method方法不存在，则返回docker命令的Help信息，若存在，调用具体的method方法，参数为args[1]及其之后所有的请求参数。

还是以一个具体的docker命令为例，docker--daemon=false--version=false pull Image_Name。通过以上Docker Client的分析，可以总结出以下执行流程。

1）解析flag参数之后，Docker将docker请求参数"pull"和"Image_Name"存放于flag.Args（）。

2）创建好的Docker Client为cli，cli执行cli.Cmd（flag.Args（）…）。

3）Cmd函数中，通过args[0]也就是"pull"，执行cli.getMethod（args[0]），获取method的名称。

4）在getMothod方法中，Docker通过处理最终返回method值为"CmdPull"。

5）最终执行method（ args[1：]...）也就是CmdPull（ args[1：]...）。

2.3.2　Docker Client执行请求命令

上一小节通过一系列的命令解析，最终找到了具体的命令执行方法，本小节主要介绍Docker Client如何通过具体的执行方法，处理并发送请求。

不同的Docker尽管请求内容不同，但是请求执行流程大致相同，故本节依旧以一个例子来阐述其中的流程，例子为：docker pull Image_Name。该命令的作用为：DockerClient发起下载镜像的请求，最终由Docker Server接收请求，由Docker Daemon完成镜像的下载与存储。

Docker Client在执行docker pull Image_Name请求命令时，执行CmdPull函数，传入参数为args[1：]...，即Image_Name。源码实现位于./docker/api/client/command.go#L1183-L1224。

下面逐一分析CmdPull的源码实现。

```
cmd := cli.Subcmd("pull", "NAME[:TAG]", "Pull an image or a repository from the
registry")
```

通过cli包中的Subcmd方法定义一个类型为Flagset的对象cmd。

```
tag := cmd.String([]string{"#t", "#-tag"}, "", "Download tagged image in a
repository")
```

为cmd对象定义一个类型为String的flag，名为"#t"或"#-tag"，初始值为空，目前这个flag参数基本已经被弃用。

```
if err := cmd.Parse(args); err != nil {
          return nil
}
```

将args参数进行第二次flag参数解析，解析过程中，先提取出是否有符合tag这个flag的参数。若有，将其赋值给tag参数，其余的参数存入cmd.NArg( )；若没有，则将所有的参数存入cmd.NArg( )中。

```
if cmd.NArg() != 1 {
     cmd.Usage()
     return nil
}
```

判断经过flag解析后的参数列表，若参数列表中参数的个数不为1，则说明需要下拉多个镜像或者没有指定下拉镜像名称，pull命令均不支持，则调用错误处理方法cmd.Usage( )，并返回nil。

```
var (
     v      = url.Values{}
     remote = cmd.Arg(0)
)
v.Set("fromImage", remote)
if *tag == "" {
     v.Set("tag", *tag)
}
```

创建一个map类型的变量v，该变量用于存放下拉镜像时所需的URL参数；随后将参数列表的第一个值cmd.Arg（0）赋给remote变量，并将remote作为键将fromImage的值添加至v。

```
remote, _ = parsers.ParseRepositoryTag(remote)
// Resolve the Repository name from fqn to hostname + name
hostname, _, err := registry.ResolveRepositoryName(remote)
if err != nil {
	return err
}
```

通过remote变量首先得到镜像的repository名称，并赋给remote自身，随后通过解析改变后的remote，得出镜像所在的host地址，即Docker Registry的地址。若用户没有制定Docker Registry的地址，则Docker默认地址为Docker Hub地址https://index.docker.io/v1/。

```
cli.LoadConfigFile()
// Resolve the Auth config relevant for this server
authConfig := cli.configFile.ResolveAuthConfig(hostname)
```

通过cli对象获取与Docker Server通信所需要的认证配置信息。

```
pull := func(authConfig registry.AuthConfig) error {
	buf, err := json.Marshal(authConfig)
	if err != nil {
		return err
	}
	registryAuthHeader := []string{
		base64.URLEncoding.EncodeToString(buf),
	}
		return cli.stream("POST", "/images/create?"+v.Encode(), nil, cli.out,
map[string][]string{
	"X-Registry-Auth": registryAuthHeader,
	})
}
```

定义一个名为pull的函数，传入的参数类型为registry.AuthConfig，返回类型为error。函数执行块中最主要的内容为：cli.stream（……）部分。该部分具体向Docker Server发送POST请求，请求的url为"/images/create？"+v.Encode（），请求的认证信息为：map[string][]string{"X-Registry-Auth"：registryAuthHeader，}。

```
if err := pull(authConfig); err != nil {
     if strings.Contains(err.Error(), "Status 401") {
          fmt.Fprintln(cli.out, "\nPlease login prior to pull:")
          if err := cli.CmdLogin(hostname); err != nil {
               return err
          }
          authConfig := cli.configFile.ResolveAuthConfig(hostname)
          return pull(authConfig)
     }
     return err
}
```

由于上一个步骤只是定义pull函数，这一步骤具体调用执行pull函数，实现实际意义上的下载请求发送。若返回成功则表明请求完成，程序直接退出，若返回错误，则做相应的错误处理。若返回错误为401，则表示用户下载的镜像必须用户先登录，随即Docker Client转至登录环节，完成之后，继续执行pull函数，若完成则最终返回。

以上便是pull请求的全部执行过程，其他请求的执行在流程上也是大同小异。总之，请求执行过程中，大多都是将命令行中关于请求的参数进行初步处理，并添加相应的辅助信息，最终通过指定的协议

向Docker Server发送Docker Client和Docker Server约定好的API请求。

2.4 总结

本章从源码的角度分析了如何通过docker可执行文件创建Docker Client，最终发送用户请求至Docker Server。

通过学习与理解Docker Client相关的源码实现，不仅可以让用户熟练掌握Docker命令的使用，还可以使用户在特殊情况下有能力修改Docker Client的源码，使其满足自身系统的某些特殊需求，以达到定制Docker Client的目 的，最大限度地发挥Docker开源思想的价值。

第3章　启动Docker Daemon

3.1　引言

自Docker诞生以来，便引领了轻量级虚拟化容器领域的技术热潮。在这一潮流下，Google、IBM、Redhat等业界翘楚纷纷加入Docker阵营。虽然目 前Docker仍主要基于Linux平台，但是Microsoft却多次宣布对Docker的支持，从先前宣布的Azure支持Docker与Kubernetes，到如今宣布的下一代Windows Server原生态支持Docker。Microsoft的这一系列举措多少喻示着向Linux世界的妥协，当然这也不得不让世人对Docker的巨大影响力有重新的认识。

Docker的影响力不言而喻，但如果需要深入学习Docker的内部实现，最重要的就是理解Docker Daemon。在Docker架构中，Docker Client通过特定的协议与Docker Daemon进行通信，而Docker Daemon主要承载了Docker运行过程中的大部分工作。

Docker Daemon是Docker架构中运行在后台的守护进程，大致可以分为Docker Server、Engine和Job三部分。三者的关系大致如下：Docker Daemon通过Docker Server模块接收Docker

Client的请求，并在Engine中处理请求，然后根据请求类型，创建出指定的Job并运行。由于用户的请求不同，DockerDaemon会创建不同的Job来完成任务，如：用户发起镜像下载请求，DockerDaemon创建名为“pull”的Job；用户发起启动容器的请求，DockerDaemon创建名为“start”的Job……

Docker Daemon的架构如图3-1所示。

本章从源码的角度，主要分析Docker Daemon的启动流程。由于Docker Daemon和Docker Client的启动流程有很多的相似之处，故本章不再赘述Docker Daemon启动的前期工作、flag参数的解析等内容，着重分析Docker Daemon启动流程中最为重要的环节：创建Daemon过程中mainDaemon（ ）的实现。

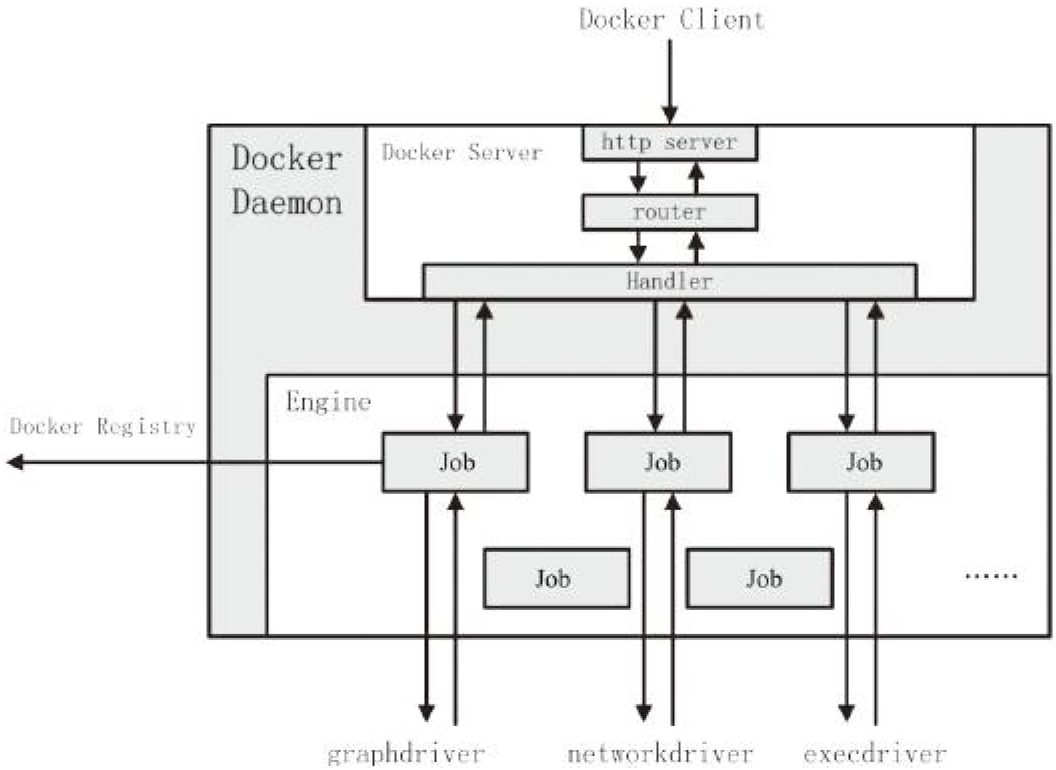


图3-1　DockerDaemon架构示意图

3.2　Docker Daemon的启动流程

Docker Daemon和Docker Client的启动均通过可执行文件docker完成，因此两者的启动流程非常相似。Docker可执行文件运行时，程序运行通过不同的命令行flag参数，区分两者，并最终运行两者各自相应的部分。

启动Docker Daemon时，一般可以使用以下命令：docker--daemon=true、docker–d；docker-d=true等。随后由Docker的main（）函数来解析以上命令的相应flag参数，并最终完成Docker Daemon的启动。

首先，附上Docker Daemon的启动流程图，如图3-2所示。

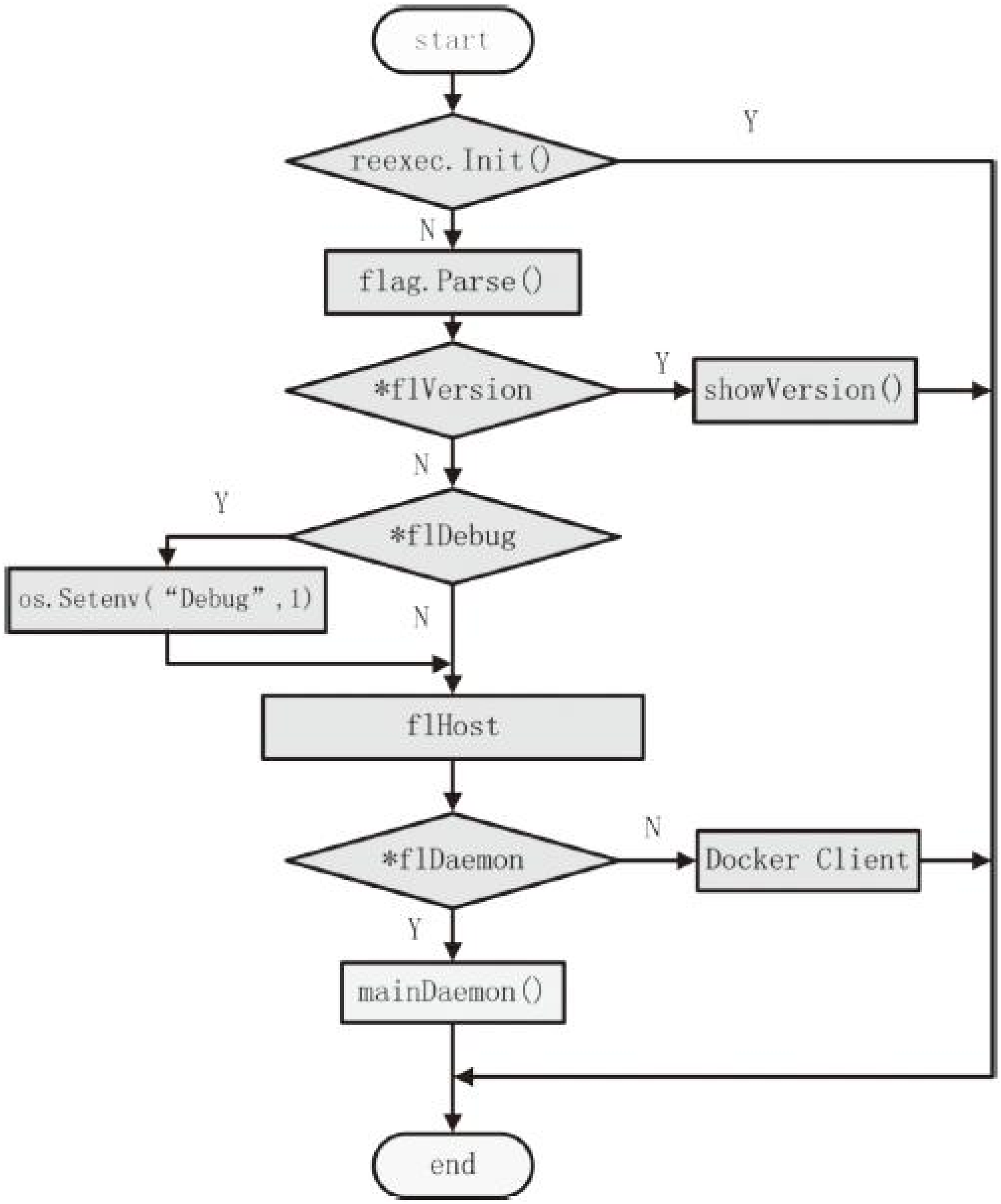


图3-2　DockerDaemon启动流程图

本书第2章已经描述了Docker中main（ ）函数运行的很多前期工作，Docker Daemon的启动也会涉及这些工作，故在此略去相同部分，主要针对后续仅和Docker Daemon相关的内容进行深入分析，即mainDaemon（ ）的具体源码实现。

3.3　mainDaemon（ ）的具体实现

Docker Daemon的启动流程图展示了DockerDaemon的从无到有。通过分析流程图，我们可以得出一个这样的结论：区分Docker Daemon与Docker Client的关键在于flag参数flDaemon的值。一旦*flDaemon的值为真，则代表docker二进制需要启动的是Docker Daemon。有关Docker Daemon的所有的工作，都被包含在函数mainDaemon（ ）的具体实现中。

宏观来讲，mainDaemon（ ）的使命是：创建一个守护进程，并保证其正常运行。

从功能的角度来说，mainDaemon（ ）实现了两部分内容：第一，创建Docker运行环境；第二，服务于Docker Client，接收并处理相应请求（完成Docker Server的初始化）。

从实现细节来分析，mainDaemon（ ）的实现流程主要包含以下步骤：

1）daemon的配置初始化。这部分在init（ ）函数中实现，即在mainDaemon（ ）运行前就执行，但由于这部分内容和mainDaemon（ ）的运行息息相关，可以认为是mainDaemon（ ）运行的先决条件。

2） 命令行flag参数检查。

3） 创建engine对象。

4） 设置engine的信号捕获及处理方法。

5） 加载builtins。

6） 使用goroutine加载daemon对象并运行。

7） 打印Docker版本及驱动信息。

8） serveapi的创建与运行。

对于以上内容，本章将一一深入分析。

3.3.1　配置初始化

mainDaemon（　）的运行位于./docker/docker/docker/daemon.go，深入分析mainDaemon（　）的实现之前，我们回到Go语言的特性，即变量与init函数的执行顺序。在daemon.go中，Docker定义了变量daemonCfg，以及init函数，通过Go语言的特性，变量的定义与init函数均会在mainDaemon（　）之前运行。两者的定义如下：

```
var (
     daemonCfg = &daemon.Config{}
)
func init() {
     daemonCfg.InstallFlags()
}
```

首先，Docker声明一个名为daemonCfg的变量，代表整个Docker Daemon的配置信息。定义完毕之后，init函数的存在，使得daemonCfg变量能获取相应的属性值。在Docker Daemon启动时，daemonCfg变量被传递至Docker Daemon并被使用。

Config对象的定义如下（含部分属性的解释），该对象位于./docker/docker/daemon/config.go：

```
type Config struct {
     Pidfile                          string  //Docker Daemon所属进程的PID文件
     Root                            string  //Docker运行时所使用的root路径
     AutoRestart                     bool    //是否一直支持创建容器的重启
```

```
	Dns                         []string//DockerDaemon为容器准备的DNS Server地址
	DnsSearch                   []string//Docker使用的指定的DNS查找地址
	Mirrors                     []string//指定的Docker Registry镜像地址
	EnableIptables              bool    //是否启用Docker的iptables功能
	EnableIpForward             bool    //是否启用net.ipv4.ip_forward功能
	EnableIpMasq                bool    //启用IP伪装技术
	DefaultIp                   net.IP  //绑定容器端口时使用的默认IP
	BridgeIface                 string  //添加容器网络至已有的网桥接口名
	BridgeIP                    string  //创建网桥的IP地址
	FixedCIDR                   string  //指定IP的IPv4子网，必须被网桥子网包含
	InterContainerCommunication bool    //是否允许宿主机上Docker容器间的通信
	GraphDriver                 string  //Docker Daemon运行时使用的特定存储驱动
	GraphOptions                []string//可设置的存储驱动选项
	ExecDriver                  string  //Docker运行时使用的特定exec驱动
	Mtu                         int     //设置容器网络接口的MTU
	DisableNetwork              bool    //是否支持Docker容器的网络模式
	EnableSelinuxSupport        bool    //是否启用对SELinux功能的支持
	Context                     map[string][]string
}
```

Docker声明daemonCfg之后，init函数实现了daemonCfg变量中各属性的赋值，具体的实现为：daemonCfg.InstallFlags（），位于./docker/docker/daemon/config.go，代码如下：

```
func (config *Config) InstallFlags() {
          flag.StringVar(&config.Pidfile, []string{"p", "-pidfile"}, 
"/var/run/docker.pid", "Path to use for daemon PID file")
          flag.StringVar(&config.Root, []string{"g", "-graph"}, 
"/var/lib/docker", "Path to use as the root of the Docker runtime")
          flag.BoolVar(&config.AutoRestart, []string{"#r", "#-restart"}, true, "-
-restart on the daemon has been deprecated infavor of --restart policies on 
docker run")
          flag.BoolVar(&config.EnableIptables, []string{"#iptables", "-
iptables"}, true, "Enable Docker's addition of iptables rules")
          flag.BoolVar(&config.EnableIpForward, []string{"#ip-forward", "-ip-
forward"}, true, "Enable net.ipv4.ip_forward")
          flag.StringVar(&config.BridgeIP, []string{"#bip", "-bip"}, "", "Use 
this CIDR notation address for the network bridge's IP, not compatible with -b")
          flag.StringVar(&config.BridgeIface, []string{"b", "-bridge"}, "", 
"Attach containers to a pre-existing network bridge\nuse 'none' to disable 
container networking")
          flag.BoolVar(&config.InterContainerCommunication, []string{"#icc", "-
icc"}, true, "Enable inter-container communication")
          flag.StringVar(&config.GraphDriver, []string{"s", "-storage-driver"}, 
"", "Force the Docker runtime to use a specific storage driver")
          flag.StringVar(&config.ExecDriver, []string{"e", "-exec-driver"}, 
"native", "Force the Docker runtime to use a specific exec driver")
          flag.BoolVar(&config.EnableSelinuxSupport, []string{"-selinux-
enabled"}, false, "Enable selinux support. SELinux does not presently support the 
BTRFS storage driver")
          flag.IntVar(&config.Mtu, []string{"#mtu", "-mtu"}, 0, "Set the 
containers network MTU\nif no value is provided: default to the default route MTU 
or 1500 if no default route is available")
```

```
        opts.IPVar(&config.DefaultIp, []string{"#ip", "-ip"}, "0.0.0.0",
"Default IP address to use when binding container ports")
        opts.ListVar(&config.GraphOptions, []string{"-storage-opt"}, "Set
storage driver options")
    // FIXME: why the inconsistency between "hosts" and "sockets"?
        opts.IPListVar(&config.Dns, []string{"#dns", "-dns"}, "Force Docker to
use specific DNS servers")
        opts.DnsSearchListVar(&config.DnsSearch, []string{"-dns-search"},
"Force Docker to use specific DNS search domains")
}
```

在函数InstallFlags（ ）的实现过程中，Docker主要定义了众多类型不一的flag参数，并将该参数的值绑定在daemonCfg变量的指定属性上，如：

```
flag.StringVar(&config.Pidfile, []string{"p", "-pidfile"}, "/var/run/docker.pid",
"Path to use for daemon PID file")
```

以上语句的含义为：

·定义一个String类型的flag参数。

·该flag的名称为"p"或者"-pidfile"。

·该flag的默认值为"/var/run/docker.pid"，并将该值绑定在变量config.Pidfile上。

·该flag的描述信息为"Path to use for daemon PID file"。

至此，关于Docker Daemon所需要的配置信息均声明并初始化完毕。

3.3.2　flag参数检查

从本小节开始，程序运行真正进入Docker Daemon的mainDaemon（），下面对此流程进行深入分析。

mainDaemon（）运行的第一个步骤是命令行flag参数的检查。具体而言，即当docker命令经过flag参数解析之后，Docker判断剩余的参数是否为0。若为0，则说明Docker Daemon的启动命令无误，正常运行；若不为0，则说明在启动Docker Daemon的时候，传入了多余的参数，此时Docker会输出错误提示，并退出运行程序。具体代码如下：

```
if flag.NArg() != 0 {
  flag.Usage()
  return
}
```

3.3.3　创建engine对象

在mainDaemon（　）运行过程中，flag参数检查完毕之后，Docker随即创建engine对象，代码如下：

```
eng := engine.New()
```

Engine是Docker架构中的运行引擎，同时也是Docker运行的核心模块。Engine扮演着Docker Container存储仓库的角色，并且通过Job的形式管理Docker运行中涉及的所有任务。

Engine结构体的定义位于./docker/docker/engine/engine.go#L47-L60，具体代码如下：

```
type Engine struct {
     handlers   map[string]Handler
     catchall   Handler
     hack       Hack // data for temporary hackery (see hack.go)
     id         string
     Stdout     io.Writer
     Stderr     io.Writer
     Stdin      io.Reader
     Logging    bool
     tasks      sync.WaitGroup
     l          sync.RWMutex // lock for shutdown
     shutdown   bool
     onShutdown []func() // shutdown handlers
}
```

Engine结构体中最为重要的是handlers属性，handlers属性为map类型，key的类型是string，value的类型是Handler。其中Handler类型的定义位于./docker/docker/engine/engine.go#L23，具体代码如下：

```
type Handler func(*Job) Status
```

可见，Handler为一个定义的函数。该函数传入的参数为Job指针，返回为Status状态。

了解完Engine以及Handler的基本知识之后，我们真正进入创建Engine实例的部分，即New（ ）函数的实现，具体代码如下：

```
func New() *Engine {
     eng := &Engine{
          handlers: make(map[string]Handler),
          id:       utils.RandomString(),
          Stdout:   os.Stdout,
          Stderr:   os.Stderr,
          Stdin:    os.Stdin,
          Logging:  true,
     }
     eng.Register("commands", func(job *Job) Status {
          for _, name := range eng.commands() {
               job.Printf("%s\n", name)
          }
          return StatusOK
     })
     // Copy existing global handlers
     for k, v := range globalHandlers {
          eng.handlers[k] = v
     }
     return eng
}
```

分析以上代码，从返回结果可以发现，New（ ）函数最终返回一个Engine实例对象。而在代码实现部分，大致可以将其分为三个步骤：

1）创建一个Engine结构体实例eng，并初始化部分属性，如handlers、id、标准输出stdout、日志属性Logging等。

2）向eng对象注册名为commands的Handler，其中Handler为临时定义的函数func（ job*Job） Status{}，该函数的作用是通过Job来打印所有已经注册完毕的command名称，最终返回状态StatusOK。

3）将变量globalHandlers中定义完毕的所有Handler都复制到eng对象的handlers属性中。

至此，一个基本的Engine对象实例eng已经创建完毕，并实现部分属性的初始化。Docker Daemon启动的后续过程中，仍然会对Engine对象实例进行额外的配置。

3.3.4　设置engine的信号捕获

Docker在包engine中执行完Engine对象的创建与初始化之后，回到mainDaemon（　）函数的运行，紧接着执行的代码为：

```
signal.Trap(eng.Shutdown)
```

Docker Daemon作为Linux操作系统上的一个后台进程，原则上应该具备处理信号的能力。信号处理能力的存在，能保障Docker管理员可以通过向Docker Daemon发送信号的方式，管理Docker Daemon的运行。

再来看以上代码则不难理解其中的含义：在Docker Daemon的运行中，设置捕获特定信号后的处理方法，特定信号有SIGINT、SIGTERM以及SIGQUIT；当程序捕获SIGINT或者SIGTERM信号时，执行相应的善后操作，最后保证Docker Daemon程序退出。

该部分代码的实现位于./docker/docker/pkg/signal/trap.go。实现的流程分为以下4个步骤：

1）创建并设置一个channel，用于发送信号通知。

2） 定义signals数组变量，初始值为os.SIGINT，os.SIGTERM；若环境变量DEBUG为空，则添加os.SIGQUIT至signals数组。

3） 通过gosignal.Notify（ c，signals...）中Notify函数来实现将接收到的signal信号传递给c。需要注意的是只有signals中被罗列出的信号才会被传递给c，其余信号会被直接忽略。

4） 创建一个goroutine来处理具体的signal信号，当信号类型为os.Interrupt或者syscall.SIGTERM时，执行传入Trap函数的具体执行方法，形参为cleanup（ ），实参为eng.Shutdown。

Shutdown（ ）函数的定义位于./docker/docker/engine/engine.go#L153-L199，主要完成的任务是：Docker Daemon关闭时，做一些必要的善后工作。

善后工作主要有以下4项：

·Docker Daemon不再接受任何新的Job。

·Docker Daemon等待所有存活的Job执行完毕。

·Docker Daemon调用所有shutdown的处理方法。

·在15秒时间内，若所有的handler执行完毕，则Shutdown（ ）函数返回，否则强制返回。

由于在signal.Trap（ eng.Shutdown） 函数的具体实现中，一旦程序接收到相应的信号，则会执行eng.Shutdown这个函数，在执行完eng.Shutdown之后，随即执行os.Exit（ 0） ，完成当前整个Docker Daemon程序的退出。源码实现位于./docker/docker/pkg/signal/trap.go#L33-L47。

3.3.5 加载builtins

DockerDaemon设置完Trap特定信号的处理方法(即eng.shutdown( ) 函数) 之后，Docker Daemon实现了builtins的加载。Docker的builtins可以理解为：Docker Daemon运行过程中，注册的一些任务(Job) ，这部分任务一般与容器的运行无关，与Docker Daemon的运行时信息有关。加载builtins的源码实现如下：

```
if err := builtins.Register(eng); err != nil {
     log.Fatal(err)
}
```

加载builtins完成的具体工作是：向engine注册多个Handler，以便后续在执行相应任务时，运行指定的Handler。这些Handler包括：Docker Daemon宿主机的网络初始化、Web API服务、事件查询、版本查看、Docker Registry的验证与搜索等。源码实现位于./docker/docker/builtins/builtins.go#L16-L30，如下：

```
func Register(eng *engine.Engine) error {
     if err := daemon(eng); err != nil {
          return err
     }
     if err := remote(eng); err != nil {
          return err
     }
     if err := events.New().Install(eng); err != nil {
          return err
     }
     if err := eng.Register("version", dockerVersion); err != nil {
```

```
            return err
        }
        return registry.NewService().Install(eng)
}
```

下面分析Register函数实现过程中最为主要的5个部分：daemon（eng）、remote（eng）、events.New（）.Install（eng）、eng.Register（"version"，dockerVersion）以及registry.NewService（）.Install（eng）。

1.注册网络初始化处理方法

daemon（eng）的实现过程，主要为eng对象注册了一个键为"init_networkdriver"的处理方法，此处理方法的值为bridge.InitDriver函数，源码如下：

```
func daemon(eng *engine.Engine) error {
    return eng.Register("init_networkdriver", bridge.InitDriver)
}
```

需要注意的是，向eng对象注册处理方法，并不代表处理方法的值函数会被立即调用执行，如注册init_networkdrive时bridge.InitDriver并不会直接运行，而是将bridge.InitDriver的函数入口作为init_networkdriver的值，写入eng的handlers属性中。当Docker Daemon接收到名为init_networkdriver的Job的执行请求时，bridge.InitDriver才被Docker Daemon调用执行。

Bridge.InitDriver的具体实现位于./docker/docker/daemon/networkdriver/bridge/driver.go#79-L175，主要作用为：

·获取为Docker服务的网络设备地址。

·创建指定IP地址的网桥。

·配置网络iptables规则。

·另外还为eng对象注册了多个Handler，如allocate_interface、release_interface、allocate_port以及link等。

本书将在第6章详细分析Docker Daemon如何初始化宿主机的网络环境。

2.注册API服务处理方法

remote（eng）的实现过程，主要为eng对象注册了两个Handler，分别为serveapi与acceptconnections，源码实现如下：

```
func remote(eng *engine.Engine) error {
     if err := eng.Register("serveapi", apiserver.ServeApi); err != nil {
          return err
     }
     return eng.Register("acceptconnections", apiserver.AcceptConnections)
}
```

注册的两个处理方法名称分别为serveapi与acceptconnections，相应的执行方法分别为apiserver.ServeApi与apiserver.AcceptConnections，具体实现位于./docker/docker/api/server/server.go。其中，ServeApi执行时，通过循环多种指定协议，创建出goroutine协调来配置指定的http.Server，最终为不同协议的请求服务；而AcceptConnections的作用主要是：通知宿主机上init守护进程Docker Daemon已经启动完毕，可以让Docker Daemon开始服务API请求。

3.注册events事件处理方法

events.New（）.Install（eng）的实现过程，为Docker注册了多个event事件，功能是给Docker用户提供API，使得用户可以通过这些API查看Docker内部的events信息，log信息以及subscribers_count信息。具体的源码位于./docker/docker/events/events.go#L29-L42，如下所示：

```
func (e *Events) Install(eng *engine.Engine) error {
	jobs := map[string]engine.Handler{
		"events":            e.Get,
		"log":               e.Log,
		"subscribers_count": e.SubscribersCount,
	}
	for name, job := range jobs {
		if err := eng.Register(name, job); err != nil {
			return err
		}
	}
	return nil
}
```

4.注册版本处理方法

eng.Register（"version"，dockerVersion）的实现过程，向eng对象注册key为version，value为dockerVersion执行方法的Handler。dockerVersion的执行过程中，会向名为version的Job的标准输出中写入Docker的版本、Docker API的版本、git版本、Go语言运行时版本，以及操作系统版本等信息。dockerVersion的源码实现如下：

```
func dockerVersion(job *engine.Job) engine.Status {
     v := &engine.Env{}
     v.SetJson("Version", dockerversion.VERSION)
     v.SetJson("ApiVersion", api.APIVERSION)
     v.Set("GitCommit", dockerversion.GITCOMMIT)
     v.Set("GoVersion", runtime.Version())
     v.Set("Os", runtime.GOOS)
     v.Set("Arch", runtime.GOARCH)
     if kernelVersion, err := kernel.GetKernelVersion(); err == nil {
          v.Set("KernelVersion", kernelVersion.String())
     }
     if _, err := v.WriteTo(job.Stdout); err != nil {
          return job.Error(err)
     }
     return engine.StatusOK
}
```

5.注册registry处理方法

registry.NewService（）.Install（eng）的实现过程位于./docker/docker/registry/service.go，功能是：在eng对象对外暴露的API信息中添加docker registry的信息。若registry.NewService（）被成功安装，则会有两个相应的处理方法注册至eng，Docker Daemon通过Docker Client提供的认证信息

向registry发起认证请求；search，在公有registry上搜索指定的镜像，目前公有的registry只支持Docker Hub。

Install的具体实现如下：

```
func (s *Service) Install(eng *engine.Engine) error {
     eng.Register("auth", s.Auth)
     eng.Register("search", s.Search)
     return nil
}
```

至此，Docker Daemon所有builtins的加载全部完成，实现了向eng对象注册特定的处理方法。

3.3.6　使用goroutine加载daemon对象并运行

Docker执行完builtins的加载之后，再次回到mainDaemon（）的执行流程中。此时，Docker通过一个goroutine协程加载daemon对象并开始运行Docker Server。这一环节的执行，主要包含以下三个步骤：

1）通过init函数中初始化的daemonCfg与eng对象，创建一个daemon对象d。

2）通过daemon对象的Install函数，向eng对象中注册众多的处理方法。

3）在Docker Daemon启动完毕之后，运行名为acceptconnections的Job，主要工作为向init守护进程发送READY=1信号，以便Docker Server开始正常接收请求。

源码实现位于./docker/docker/docker/daemon.go#L43-L56，如下所示：

```
go func() {
     d, err := daemon.MainDaemon(daemonCfg, eng)
     if err != nil {
          log.Fatal(err)
     }
     if err := d.Install(eng); err != nil {
          log.Fatal(err)
     }
```

```
        if err := eng.Job("acceptconnections").Run(); err != nil {
            log.Fatal(err)
        }
    }()
```

下面详细分析三个步骤所做的工作。

1.创建daemon对象

daemon.NewDaemon（daemonCfg，eng）是创建daemon对象d的核心部分，主要作用是初始化Docker Daemon的基本环境，如处理config参数，验证系统支持度，配置Docker工作目录，设置与加载多种驱动，创建graph环境，验证DNS配置等。

由于daemon.MainDaemon（daemonCfg，eng）是加载Docker Daemon的核心部分，且篇幅过长，本书第4章将深入分析NewDaemon的实现。

2.通过daemon对象为engine注册Handler

Docker创建完daemon对象，goroutine立即执行d.Install（eng），具体实现位于./docker/daemon/daemon.go，代码如下所示：

```
    func (daemon *Daemon) Install(eng *engine.Engine) error {
        for name, method := range map[string]engine.Handler{
            "attach":            daemon.ContainerAttach,
            "build":             daemon.CmdBuild,
            "commit":           daemon.ContainerCommit,
            "container_changes":  daemon.ContainerChanges,
            "container_copy":     daemon.ContainerCopy,
            "container_inspect":   daemon.ContainerInspect,
```

```
            "containers":         daemon.Containers,
            "create":            daemon.ContainerCreate,
            "delete":            daemon.ContainerDestroy,
            "export":            daemon.ContainerExport,
            "info":              daemon.CmdInfo,
            "kill":               daemon.ContainerKill,
            ...
            "image_delete":      daemon.ImageDelete,
        } {
            if err := eng.Register(name, method); err != nil {
                return err
            }
        }
        if err := daemon.Repositories().Install(eng); err != nil {
            return err
        }
        eng.Hack_SetGlobalVar("httpapi.daemon", daemon)
        return nil
    }
```

以上代码的实现同样分为三部分：

·向eng对象中注册众多的处理方法对象。

·daemon.Repositories（ ）.Install（ eng）实现了向eng对象注册多个与Docker镜像相关的Handler，Install的实现位于./docker/docker/graph/service.go。

·eng.Hack_SetGlobalVar（ "httpapi.daemon"，daemon）实现向eng对象中类型为map的hack对象中添加一条记录，键为httpapi.daemon，值为daemon。

3.运行名为acceptconnections的Job

Docker在goroutine的最后环节运行名为acceptconnections的Job，主要作用是通知init守护进程，使Docker Daemon开始接受

请求。源码位于./docker/docker/docker/daemon.go#L53-L55，如下所示：

```
// after the daemon is done setting up we can tell the api to start
// accepting connections
if err := eng.Job("acceptconnections").Run(); err != nil {
     log.Fatal(err)
}
```

关于Job的运行流程大同小异，总结而言，都是首先创建特定名称的Job，其次为Job配置环境参数，最后运行Job对应Handler的函数。作为本书涉及的第一个具体Job，下面将对acceptconnections这个Job的执行进程深入分析。

eng.Job（"acceptconnections"）.Run（）的运行包含两部分：首先执行eng.Job（"acceptconnections"），返回一个Job实例，随后再执行该Job实例的Run（）函数。

eng.Job（"acceptconnections"）的实现位于./docker/docker/engine/engine.go#L115-L137，如下所示：

```
func (eng *Engine) Job(name string, args ...string) *Job {
     job := &Job{
          Eng:    eng,
          Name:   name,
          Args:   args,
          Stdin:  NewInput(),
          Stdout: NewOutput(),
          Stderr: NewOutput(),
          env:    &Env{},
     }
     if eng.Logging {
          job.Stderr.Add(utils.NopWriteCloser(eng.Stderr))
     }
     if handler, exists := eng.handlers[name]; exists {
```

```
        job.handler = handler
    } else if eng.catchall != nil && name != "" {
        job.handler = eng.catchall
    }
    return job
}
```

通过分析以上创建Job的源码，我们可以发现Docker首先创建一个类型为Job的job对象，该对象中Eng属性为函数的调用者eng，该对象的Name属性为acceptconnections，没有其他参数传入。另外在eng对象所有的handlers属性中寻找key为acceptconnections所对应的value值（即具体的Handler）。由于在加载builtins时，源码remote（eng）已经向eng注册过这样一条记录，键为acceptconnections，值为apiserver.AcceptConnections。因此，Job对象的handler属性为apiserver.AcceptConnections。最后函数返回已经初始化完毕的对象Job。

创建完Job对象之后，随即执行该job对象的run（）函数。run（）函数的源码实现位于./docker/docker/engine/job.go#L48-L96，该函数执行指定的Job，并在Job执行完成前一直处于阻塞状态。对于名为acceptconnections的Job对象，运行代码为job.status= job.handler（job），由于job.handler值为apiserver.AcceptConnections，故真正执行的是job.status= apiserver.AcceptConnections（job）。

AcceptConnections的具体实现属于Docker Server的范畴，深入研究Docker Server可以发现，这部分源码位于./docker/docker/api/server/server.go#L1370-L1380，如下所示：

```
func AcceptConnections(job *engine.Job) engine.Status {
     // Tell the init daemon we are accepting requests
     go systemd.SdNotify("READY=1")
     if activationLock != nil {
          close(activationLock)
     }
     return engine.StatusOK
}
```

AcceptConnections函数的重点是go systemd.SdNotify（"READY=1"）的实现，位于./docker/docker/pkg/system/sdnotify.go#L12-L33，主要作用是通知init守护进程Docker Daemon的启动已经全部完成，潜在的功能是要求Docker Daemon开始接收并服务Docker Client发送来的API请求。

至此，通过goroutine来加载daemon对象并运行启动Docker Server的工作全部完成。

3.3.7 打印Docker版本及驱动信息

Docker再次回到mainDaemon（ ）的运行流程，由于Go语言goroutine的性质，在goroutine执行之时，mainDaemon（ ）函数内部其他代码也会并发执行。

第一个执行的即为显示Docker的版本信息、GitCommit信息、ExecDriver和GraphDriver这两个驱动的具体信息，源码如下：

```
log.Printf("docker daemon: %s %s; execdriver: %s; graphdriver: %s",
     dockerversion.VERSION,
     dockerversion.GITCOMMIT,
     daemonCfg.ExecDriver,
     daemonCfg.GraphDriver,
)
```

3.3.8 serveapi的创建与运行

打印Docker的部分具体信息之后，Docker Daemon立即创建并运行名为serveapi的Job，主要作用为让Docker Daemon提供Docker Client发起的API服务。实现代码位于./docker/docker/docker/daemon.go#L66，如下所示：

```
job := eng.Job("serveapi", flHosts...)
job.SetenvBool("Logging", true)
job.SetenvBool("EnableCors", *flEnableCors)
job.Setenv("Version", dockerversion.VERSION)
job.Setenv("SocketGroup", *flSocketGroup)
job.SetenvBool("Tls", *flTls)
job.SetenvBool("TlsVerify", *flTlsVerify)
job.Setenv("TlsCa", *flCa)
job.Setenv("TlsCert", *flCert)
job.Setenv("TlsKey", *flKey)
job.SetenvBool("BufferRequests", true)
if err := job.Run(); err != nil {
     log.Fatal(err)
}
```

以上代码标志着Docker Daemon真正进入状态。实现过程中，Docker首先创建一个名为serveapi的Job，并将flHosts的值赋给job.Args。flHosts的作用主要是：为Docker Daemon提供使用的协议与监听的地址。随后，Docker Daemon为该Job设置了众多的环境变量，如安全传输层协议的环境变量等。最后通过job.Run( )运行该serveapi的Job。

由于在eng中key为serveapi的handler，value为apiserver.ServeApi，故该Job运行时，执行apiserver.ServeApi函数，位于./docker/docker/api/server/server.go。ServeApi函数的作用是：对于所有用户定义支持协议，Docker Daemon均创建一个goroutine来启动相应的http.Server，并为每一种协议服务。

由于创建并启动http.Server为Docker架构中有关Docker Server的重要内容，本书将在第5章深入介绍Docker Server。

至此，我们可以认为Docker Daemon已经完成了serveapi这个Job的初始化工作。一旦acceptconnections这个Job运行完毕，Docker Daemon则会通知init进程Docker Daemon启动完毕，可以开始提供API服务，两个Job通过activationLock进行同步。

3.4 总结

本章从源码的角度分析了Docker Daemon的启动，着重分析了mainDaemon（ ）的实现。

Docker Daemon作为Docker架构中的主干部分，负责了Docker内部几乎所有操作的管理。学习Docker Daemon的具体实现，可以对Docker架构有一个较为全面的认识。总结而言，Docker的运行载体为Daemon，调度管理由Engine负责，任务执行靠Job。

第4章　Docker Daemon之NewDaemon实现

4.1　引言

Docker的生态系统日趋完善，开发者群体也在日趋庞大，这些现象都使得工业界对Docker持续抱有乐观的态度。如今，对于广大开发者而言，使用Docker已然不是门槛，享受Docker带来的福利也不再困难。然而，如何探寻Docker适应的场景，如何发展Docker周边的技术，以及如何弥合Docker新技术与传统物理机或虚拟机技术之间的鸿沟，已经成为Docker研究者们要思考与解决的问题。

在Docker架构中Docker Daemon支撑着整个后台进程的运行，同时也统一化管理着Docker架构中graph、graphdriver、execdriver、volumes、Docke容器等众多资源。可以说，Docker Daemon复杂的运作均由daemon对象来调度，而NewDaemon的实现恰巧可以帮助大家了解这一切的来龙去脉。通过本章内容的介绍，我们力求帮助广大Docker爱好者更多地理解Docker的核心——Docker Daemon的实现。

本章从源码角度，分析Docker Daemon加载过程中NewDaemon的实现，整个分析过程如图4-1所示。

由图4-1可见，Docker Daemon中NewDaemon的执行流程主要包含12个独立的步骤：处理配置信息、检测系统支持及用户权限、配置工作路径、加载并配置graphdriver、创建Docker Daemon网络环境、创建并初始化graphdb、创建execdriver、创建daemon实例、检测DNS配置、加载已有容器、设置shutdown处理方法，以及返回daemon实例。

下面我们将在NewDaemon的具体实现中，详细分析以上内容。

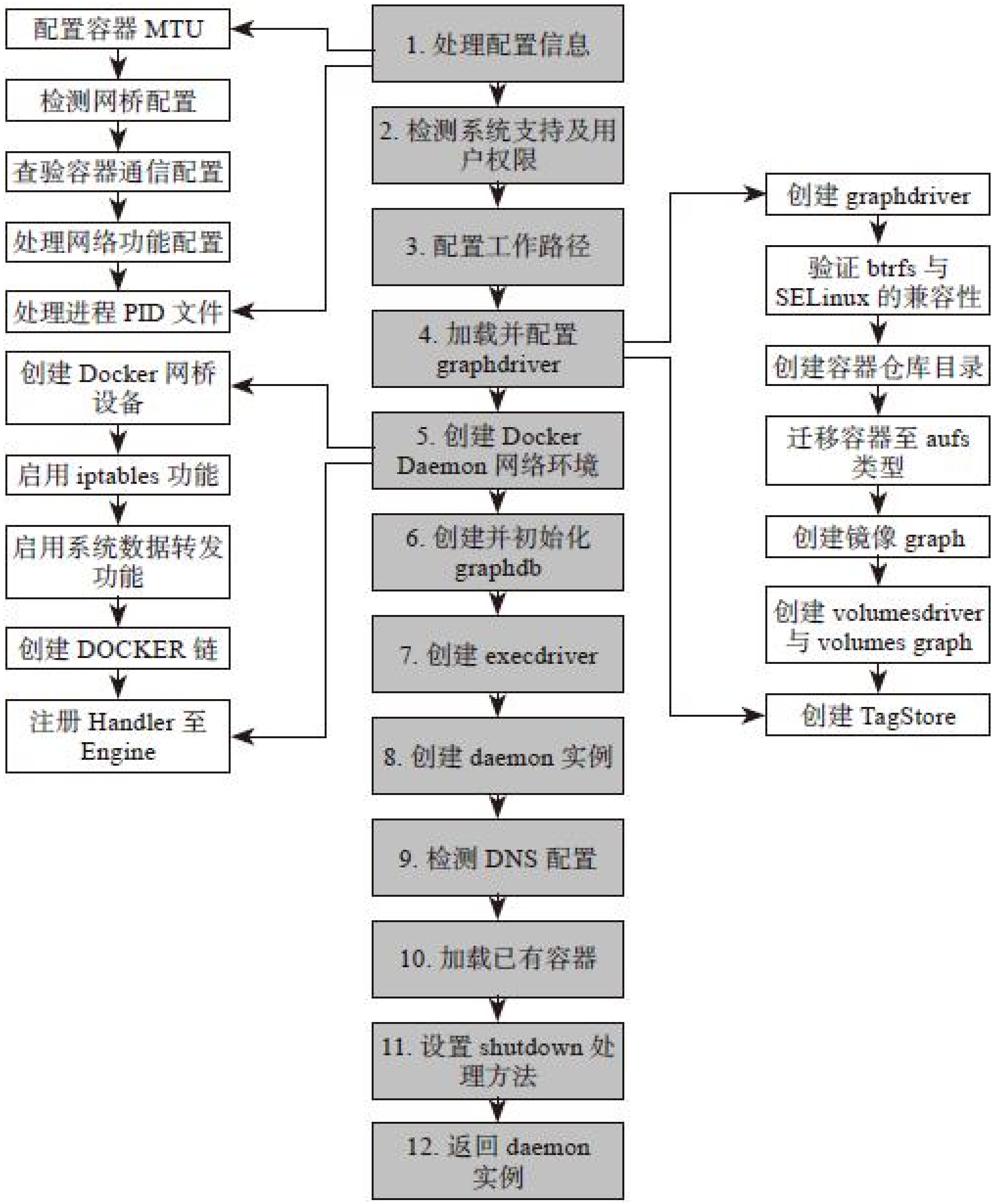


图4-1　Docker Daemon中NewDaemon执行流程图

4.2 NewDaemon具体实现

本书第3章对于Docker Daemon启动的分析过程，有这样一个环节：使用协程加载daemon对象。在加载并运行daemon对象时，所做的第一个工作是：

```
d, err := daemon.NewDaemon(daemonCfg, eng)
```

简单分析一下这行代码。

·函数名：NewDaemon

·调用此函数的包名：daemon

·函数具体实现源文件：./docker/docker/daemon/daemon.go
·函数传入实参：daemonCfg，定义Docker Daemon运行过程中所需的众多配置信息；eng，在mainDaemon中创建的Engine对象实例

·函数返回内容：d，具体的Daemon对象实例；err，错误状态

进入./docker/docker/daemon/daemon.go中，寻找函数NewDaemon的具体实现，源码如下：

```
func NewDaemon(config *Config, eng *engine.Engine) (*Daemon, error) {
	daemon, err := NewDaemonFromDirectory(config, eng)
	if err != nil {
		return nil, err
	}
	return daemon, nil
}
```

可见，在实现NewDaemon的过程中，要通过NewDaemonFromDirectory函数来实现创建Daemon的运行环境。该函数的实现，传入参数以及返回类型与NewDaemon函数完全相同。接下来将详细分析NewDaemonFromDirectory的实现细节。

4.3 应用配置信息

NewDaemonFromDirectory的实现过程中，第一个工作是：如何应用传入的配置信息。这部分配置信息服务于Docker Daemon的运行，并在Docker Daemon启动初期初始化完毕。配置信息的主要功能是：供用户自由配置Docker的可选功能，使得Docker的运行更贴近用户期待的运行场景。

配置信息的处理包含4部分：

·配置Docker容器的MTU

·检测网桥配置信息

·查验容器间的通信配置

·处理PID文件配置

下面逐一分析配置信息的处理。

4.3.1　配置Docker容器的MTU

config信息中的Mtu属性应用于容器网络接口的最大传输单元（MTU）特性。有关MTU的源码如下：

```
if config.Mtu == 0 {
          config.Mtu = GetDefaultNetworkMtu()
}
```

可见，若config信息中Mtu的值为0，Docker则通过GetDefaultNetworkMtu函数将Mtu设定为默认的值；否则，采用config中的Mtu值。由于在默认的配置文件./docker/docker/daemon/config.go（下文简称为默认配置文件）中，Mtu属性的初始值为0，故执行GetDefaultNetworkMtu。

GetDefaultNetworkMtu函数的具体实现位于./docker/daemon/config.go#L65-L70，如下：

```
func GetDefaultNetworkMtu() int {
     if iface, err := networkdriver.GetDefaultRouteIface(); err == nil {
          return iface.MTU
     }
     return defaultNetworkMtu
}
```

在GetDefaultNetworkMtu的实现中，Docker通过networkdriver包的GetDefaultRouteIface方法获取具体的网络接

口，若该网络接口存在，则返回该网络接口的MTU属性值；否则返回默认的MTU值defaultNetworkMtu，值为1500。

4.3.2　检测网桥配置信息

处理完config中的Mtu属性之后，Docker马上检测config信息中BridgeIface和BridgeIP这两个属性。BridgeIface和BridgeIP的作用是为创建网桥的任务init_networkdriver提供参数，源码如下：

```
if config.BridgeIface != "" && config.BridgeIP != "" {
           return nil, fmt.Errorf("You specified -b & --bip, mutually exclusive
options. Please specify only one.")
}
```

以上代码的含义为：若config中BridgeIface和BridgeIP两个属性均不为空，则返回nil对象，并返回错误信息，错误信息内容为：用户同时指定了BridgeIface和BridgeIP，这两个属性属于互斥类型，只能指定其中之一。原因是：当用户为Docker网桥选定已经存在的网桥接口时，应该沿用已有网桥的当前IP地址，不应再提供IP地址；当用户不选已经存在的网桥接口作为Docker网桥时，Docker会另行创建一个全新的网桥接口作为Docker网桥，此时用户可以为此新创建的网桥接口设定自定义IP地址；当然两者都不选的话，Docker会为用户接管完整的Docker网桥创建流程，从创建默认的网桥接口，到尾网桥接口设置默认的IP地址。而在默认配置文件中，BridgeIface和BridgeIP的值均为空字符串。

4.3.3 查验容器间的通信配置

查验容器间的通信配置，主要是针对config信息中的EnableIptables和Inter-Container Communication属性。

EnableIptables属性主要来源于flag参数--iptables，它的作用是：在DockerDaemon启动时，是否对宿主机的iptables规则作修改。若EnableIptables的值为false，则代表Docker Daemon启动时不对宿主机的iptables规则作任何修改；若EnableIptables的值为true，则代表Docker Daemon启动时对宿主机的iptables规则作修改。

仅仅分析EnableIptables的作用，显得有些抽象，理解起来也较为生涩。结合InterContainerCommunication来分析，就会变得自然很多。InterContainerCommunication属性来源于flag参数--icc，它的作用是：在Docker Daemon启动时，是否开启Docker容器之间互相通信的功能。若InterContainerCommunication的值为false，则Docker Daemon会在宿主机iptables的FORWARD链中添加一条Docker容器间流量均DROP的规则；若InterContainerCommunication为true，则Docker Daemon会在

宿主机iptables的FORWARD链中添加一条Docker容器间流量均ACCEPT的规则。

通过分析以上的内容，我们可以发现：InterContainerCommunication的值不论为false还是为true，都会在Docker Daemon启动时，动用iptables，故EnableIptables的值只能为true，必须允许Docker Daemon启动时对iptables规则作修改的操作。另外，若EnableIptables为true，则说明Docker Daemon启动时允许对iptables规则作修改，而此时若InterContainerCommunication为false，则说明Docker Daemon添加一条DROP的规则，导致Docker容器间不能互相通信，在此情况下，Docker提供容器间link的机制，仍然可以帮助容器实现互相通信。

查验容器间通信配置的源码如下：

```
if !config.EnableIptables && !config.InterContainerCommunication {
          return nil, fmt.Errorf("You specified --iptables=false with --
icc=false. ICC uses iptables to function. Please set --icc or --iptables to
true.")
}
```

代码含义为：若EnableIptables和InterContainerCommunication两个属性的值均为false，则返回nil对象以及错误信息。其中错误信息为：用户将以上两属性均置为

false，容器间通信需要iptables的支持，需设置其中之一为true。而在默认配置文件中，这两个属性的值均为true。

4.3.4 处理网络功能配置

接着，Docker处理config中的DisableNetwork属性，后续创建并执行创建Docker Daemon网络环境时会使用此属性，即在名为init_networkdriver的Job创建并运行中体现。

```
config.DisableNetwork = config.BridgeIface == DisableNetworkBridge
```

由于config信息中的BridgeIface属性值为空，另外DisableNetworkBridge的值为字符串none，因此最终config中DisableNetwork的值为false。后续名为init_networkdriver的Job执行时，需要使用DisableNetwork这个属性。

4.3.5　处理PID文件配置

处理PID文件配置，主要工作是：为运行时Docker Daemon进程的PID号创建一个PID文件，文件的路径即为config中的Pidfile属性，并且为Docker Daemon的shutdown操作添加一个删除此Pidfile的函数，以便在Docker Daemon退出的时候，可以在第一时间删除Pidfile。实现处理PID文件配置信息的源码如下：

```
if config.Pidfile != "" {
        if err := utils.CreatePidFile(config.Pidfile); err != nil {
                return nil, err
        }
        eng.OnShutdown(func() {
                utils.RemovePidFile(config.Pidfile)
        })
}
```

在代码执行过程中，首先检测config中的Pidfile属性是否为空，若为空，则跳过代码块继续执行；若不为空，则首先在文件系统中创建具体的Pidfile，然后向eng的onShutdown属性添加一个处理函数，函数具体完成的工作为utils.RemovePidFile（config.Pidfile），即在Docker Daemon进行shutdown操作的时候，删除Pidfile文件。在默认配置文件中，Pidfile文件的初始值为"/var/run/docker.pid"。

以上便是关于NewDaemon实现过程中配置信息的处理分析。

4.4　检测系统支持及用户权限

初步处理完Docker的配置信息之后，Docker立即对自身的运行环境进行一系列的检测。检测主要包括以下三方面：

·操作系统类型对Docker Daemon的支持；

·用户权限的级别；

·内核版本与处理器的支持。

系统支持与用户权限检测的实现较为简单，源码实现如下：

```
if runtime.GOOS != "linux" {
      log.Fatalf("The Docker daemon is only supported on linux")
}
if os.Geteuid() != 0 {
      log.Fatalf("The Docker daemon needs to be run as root")
}
if err := checkKernelAndArch(); err != nil {
      log.Fatalf(err.Error())
}
```

首先，通过runtime.GOOS检测操作系统的类型。runtime.GOOS返回运行程序所在操作系统的类型，可以是Linux、Darwin、FreeBSD等。结合具体代码，可以发现，若操作系统不为Linux，将报出Fatal错误日志，内容为“Docker Daemon只能支持Linux操作系统”。

接着，通过os.Geteuid（ ），检测程序用户是否拥有足够权限。os.Geteuid（ ）返回调用者所在组的组id。结合具体源码分析可知，若返回不为0，则说明docker程序不是以root用户的身份运行，报出Fatal错误日志。

最后，通过checkKernelAndArch（ ），检测内核的版本以及主机处理器类型。checkKernel-AndArch（ ）的实现同样位于./docker/docker/daemon/daemon.go#L1097-L1119。实现过程中，第一个工作是：检测程序运行所在的处理器架构是否为“amd64”，而目前Docker运行时只能支持amd64的处理器架构。第二个工作是：检测Linux内核版本是否满足要求，而目前Docker Daemon运行所需的内核版本若过低，则很有可能出现不稳定的状况，因此Docker官方建议用户升级内核版本至3.8.0或以上版本（包括3.8.0）。

4.5 配置工作路径

配置Docker Daemon的工作路径，主要是创建Docker Daemon运行中所在的工作目录。实现过程中，通过config中的Root属性来完成。Docker Daemon的root目录作用非常大，几乎涵盖Docker在宿主机上运行的所有信息，包括：所有的Docker镜像内容、所有Docker容器的文件系统、所有Docker容器的元数据、所有容器的数据卷内容等。

在默认配置文件中，Root属性的值为”/var/lib/docker”。

在配置工作路径的代码实现中，步骤如下：

1） 使用规范路径创建一个TempDir，路径名为tmp。

2） 通过tmp，创建一个指向tmp的文件符号连接realTmp。

3） 使用realTemp的值，创建并赋值给环境变量TMPDIR。

4） 处理config的属性EnableSelinuxSupport。

5） 将realRoot重新赋值于config.Root，并创建Docker Daemon的工作根目录。

4.6 加载并配置graphdriver

加载并配置存储驱动graphdriver，目的在于：使得Docker Daemon创建Docker镜像管理所需的驱动环境。graphdriver用于完成Docker镜像的管理，包括获取、存储以及容器rootfs的构建等。

4.6.1　创建graphdriver

创建graphdriver的内容源码位于./docker/docker/daemon/daemon.go#L743-L790。具体细节分析如下：

```
graphdriver.DefaultDriver = config.GraphDriver
driver, err := graphdriver.New(config.Root, config.GraphOptions)
```

首先，Docker对graphdriver包中的DefaultDriver对象赋值，值为config中的GraphDriver属性，在默认配置文件中，GraphDriver属性的值为空；同样的，属性GraphOptions也为空。然后通过GraphDriver中的new函数实现加载graph的存储驱动。

创建具体的graphdriver是极其重要的一个环节，实现细节由graphdriver包中的New函数来完成。New函数的实现位于./docker/docker/daemon/graphdriver/driver.go#L81-L111，实现步骤如下：

第一，遍历数组选择graphdriver，数组内容为os.Getenv（"DOCKER_DRIVER"）和DefaultDriver。若数组不为空，graphdriver则通过GetDriver函数直接返回相应的Driver对象实例；若为空，则继续往下执行。这部分内容的作用是：让

graphdriver的加载首先满足用户的自定义选择，用户可以通过定义环境变量的方式定义graphdriver的类型，其次Docker采用默认值，源码如下：

```
for _, name := range []string{os.Getenv("DOCKER_DRIVER"), DefaultDriver} {
     if name != "" {
          return GetDriver(name, root, options)
     }
}
```

第二，遍历优先级数组priority选择graphdriver，优先级数组priority的内容依次为aufs、btrfs、devicemapper和vfs。若遍历验证时，GetDriver成功，则直接返回当前的Driver对象实例；若不成功，则继续往下执行。这部分内容的作用是：在没有指定以及默认的驱动时，从优先级数组中选择驱动，目前优先级最高的为aufs，源码如下：

```
for _, name := range priority {
     driver, err = GetDriver(name, root, options)
     if err != nil {
          if err == ErrNotSupported || err == ErrPrerequisites || err ==
ErrIncompatibleFS {
               continue
          }
          return nil, err
     }
     return driver, nil
}
```

第三，从已经注册的drivers数组中选择graphdriver，源码如下：

```
for _, initFunc := range drivers {
     if driver, err = initFunc(root, options); err != nil {
               if err == ErrNotSupported || err == ErrPrerequisites || err ==
ErrIncompatibleFS {
               continue
          }
               return nil, err
     }
     return driver, nil
}
return nil, fmt.Errorf("No supported storage backend found")
```

在aufs、btrfs、devicemapper和vfs四个不同类型驱动的init函数中，它们均向graphdriver的drivers数组注册了相应的初始化方法。分别位于./docker/docker/daemon/graphdriver/aufs/aufs.go，以及其他三类驱动的相应位置。这部分内容的作用是：在没有优先级数组的时候，同样可以通过注册的驱动来选择具体的graphdriver。

4.6.2 验证btrfs与SELinux的兼容性

由于目前在btrfs文件系统上运行的Docker不兼容SELinux，因此当config中配置信息需要启用SELinux的支持并且驱动的类型为btrfs时，返回nil对象，并报出Fatal日志。代码实现如下：

```
// As Docker on btrfs and SELinux are incompatible at present, error on both
being enabled

if config.EnableSelinuxSupport && driver.String() == "btrfs" {

     return nil, fmt.Errorf("SELinux is not supported with the BTRFS graph
driver!")

}
```

4.6.3 创建容器仓库目录

Docker Daemon在创建Docker容器之后，需要将容器的元数据信息放置于某个仓库目录下，统一管理。而这个目录即为daemonRepo，值为："/var/lib/docker/containers"。Docker通过daemonRepo创建对应的目录。源码实现如下：

```
daemonRepo := path.Join(config.Root, "containers") if err :=
os.MkdirAll(daemonRepo, 0700); err != nil && !os.IsExist(err) {


    return nil, err }
```

4.6.4 迁移容器至aufs类型

Docker Daemon的启动很有可能不是第一次，因此Docker环境中也有可能存在一部分的遗留内容；Docker Daemon也有可能是版本升级后的第一次启动，Docker环境中同样有可能存在遗留内容。因此，当graphdriver的类型为aufs时，DockerDaemon启动需要将现有容器root目录下的相应内容都迁移至aufs类型；若不为aufs，则继续往下执行。

对于aufs类型的graphdriver，在Docker 0.7.x版本之前，Docker将容器镜像的镜像层内容以及镜像元数据均放在同一个目录下，迁移操作要完成将以上两者拆分存储，以满足新版Docker的aufs支持。关于Docker镜像的存储，可参见第10章。

实现源码如下：

```
if err = migrateIfAufs(driver, config.Root); err != nil {
        return nil, err
}
```

这部分的迁移内容主要包括Repositories、Images以及Containers，具体源码实现位

于./docker/docker/daemon/graphdriver/aufs/migrate.go#L39-L50，如下所示：

```
func (a *Driver) Migrate(pth string, setupInit func(p string) error) error {
     if pathExists(path.Join(pth, "graph")) {
          if err := a.migrateRepositories(pth); err != nil {
               return err
          }
          if err := a.migrateImages(path.Join(pth, "graph")); err != nil {
               return err
          }
          return a.migrateContainers(path.Join(pth, "containers"), setupInit)
     }
     return nil
}
```

迁移镜像库的功能是：在Docker Daemon的root工作目录下创建repositories-aufs的文件，存储所有与镜像相关的镜像库以及镜像标签信息。

迁移Docker镜像的主要功能是：将原有的image镜像都迁移至aufs驱动能识别并使用的类型，主要是拆分镜像原先的存储方式，将内容迁移到aufs所规定的layers、diff与mnt目录。

迁移容器的主要功能是：将Docker原先的容器运行环境使用aufs驱动来进行迁移配置，包括创建容器的rootfs，配置容器初始层（init layer），创建容器的读写层等。

4.6.5　创建镜像graph

创建镜像graph的主要工作是：通过Docker的root目录以及graphdriver实例，实例化一个全新的graph对象，用以管理在文件系统中Docker的root路径下graph目录的内容。graph目录下的文件以镜像ID为单位，分别存储单个镜像的json文件以及镜像的大小文件layersize。其中镜像的json文件包含镜像自身的元数据信息。实现源码如下：

```
g, err := graph.NewGraph(path.Join(config.Root, "graph"), driver)
```

NewGraph的具体实现位于./docker/docker/graph/graph.go，实现过程中返回的对象为Graph类型，定义如下：

```
type Graph struct {
     Root    string
     idIndex *truncindex.TruncIndex
     driver  graphdriver.Driver
}
```

其中Root表示graph的工作根目录，一般为"/var/lib/docker/graph"；idIndex使得检索字符串标识符时，允许使用任意一个该字符串唯一的前缀，只要该前缀全局唯一，则可确

保找到相应的镜像。在这里，idIndex用于通过简短有效的字符串 前缀检索镜像的ID；最后driver表示具体的graphdriver类型。

4.6.6　创建volumesdriver以及volumes graph

在Docker中数据卷(volume) 的概念是：可以从Docker宿主机上挂载到Docker容器内部的特定目录。一个数据卷可以被多个Docker容器挂载，从而使Docker容器可以实现互相共享数据等。在实现数据卷时，Docker需要使用driver来管理它，又由于数据卷的管理不会像容器文件系统管理那么复杂，故Docker采用vfs驱动实现数据卷的管理。

Docker的范畴中，数据卷可以分为两种：第一种，用户使用docker run命令启动容器时传入-v A：B，使得宿主机上的目录A可以挂载到容器内部的目录B，；第二种，用户在使用Dockerfile时使用命令VOLUME/data，或者使用dockerrun命令启动容器时传入-v/data，用户虽然指定在宿主机上的目录，但是Docker Daemon一般情况下会接管宿主机上的目录创建，并再将目录挂载至Docker容器内部，此时宿主机上的目录一般位于"/var/lib/docker/vfs/dir/<ID>"。第一种数据卷通常可以将其称为bind-mount volume，而第二种通常称为data volume。

于volumes graph上创建volumesdriver的源码实现如下：

```
volumesDriver, err := graphdriver.GetDriver("vfs", config.Root,
config.GraphOptions)
volumes, err := graph.NewGraph(path.Join(config.Root, "volumes"), volumesDriver)
```

主要完成工作为：使用vfs这种类型的driver创建volumesDriver；在Docker的root路径下创建volumes目录，并返回volumes这个graph对象实例。

4.6.7 创建TagStore

TagStore主要是用于管理存储镜像的仓库列表（repository list）。

创建tagStore的源码如下：

```
repositories, err := graph.NewTagStore(path.Join(config.Root, "repositories-"+driver.String()), g)
```

函数NewTagStore的实现位于./docker/docker/graph/tags.go，TagStore的定义如下：

```
type TagStore struct {
     path         string
     graph        *Graph
     Repositories map[string]Repository
     sync.Mutex
     pullingPool map[string]chan struct{}
     pushingPool map[string]chan struct{}
}
```

需要阐述的是TagStore类型中的多个属性的含义。

·path：TagStore中记录镜像仓库的文件所在路径，如aufs类型的TagStore path的值为"/var/lib/docker/repositories-aufs"。

·graph：相应的Graph实例对象。

·Repositories：记录镜像仓库的映射数据结构。

·sync.Mutex：TagStore的互斥锁。

·pullingPool：记录池，记录有哪些镜像正在被下载，若某一个镜像正在被下载，则驳回其他Docker Client发起下载该镜像的请求。

·pushingPool：记录池，记录有哪些镜像正在被上传，若某一个镜像正在被上传，则驳回其他Docker Client发起上传该镜像的请求。

4.7 配置Docker Daemon网络环境

创建Docker Daemon运行环境的时候，配置Docker所在宿主机的网络环境是极为重要的一个环节。这不仅关系着将来容器对外的通信，同样也关系着容器间的通信。

配置Docker宿主机的网络环境时，Docker Daemon通过运行名为init_networkdriver的Job来完成。源码实现如下：

```
if !config.DisableNetwork {
     job := eng.Job("init_networkdriver")
     job.SetenvBool("EnableIptables", config.EnableIptables)
     job.SetenvBool("InterContainerCommunication",
config.InterContainerCommunication)
     job.SetenvBool("EnableIpForward", config.EnableIpForward)
     job.Setenv("BridgeIface", config.BridgeIface)
     job.Setenv("BridgeIP", config.BridgeIP)
     job.Setenv("DefaultBindingIP", config.DefaultIp.String())
     if err := job.Run(); err != nil {
          return nil, err
     }
}
```

分析以上源码可知，通过config中的DisableNetwork属性来判断是否执行Job。在默认配置文件中，该属性曾 被定义，却没有初始值，然而在应用配置信息这个步骤(4.3节) ，Docker处理网络功能配置时，将DisableNetwork属性赋值为false。故以上判断语句结果为true，执行相应的代码块。

配置DockerDaemon网络环境的工作主要通过init_networkdriver这个Job来完成。Docker首先创建名为init_networkdriver的Job，随后为此Job设置环境变量，环境变量的值如下：

·环境变量EnableIptables，使用config.EnableIptables来赋值，默认值为true。

·环境变量InterContainerCommunication，使用config.InterContainerCommunication来赋值，为默认值true。

·环境变量EnableIpForward，使用config.EnableIpForward来赋值，默认值为true。

·环境变量BridgeIface，使用config.BridgeIface来赋值，为空字符串""。

·环境变量BridgeIP，使用config.BridgeIP来赋值，为空字符串""。

·环境变量DefaultBindingIP，使用config.DefaultIp.String（）来赋值，默认值为"0.0.0.0"。

设置完环境变量之后，Docker随即运行此Job，由于在eng中key为init_networkdriver的handler，value为bridge.InitDriver

函数，故执行bridge.InitDriver函数，具体的实现位于./docker/docker/daemon/networkdriver/bridge/dirver.go，作用为：

·获取为Docker容器服务的网络接口IP地址。

·创建指定IP地址的网桥接口。

·启用Iptables功能并进行配置。

·另外，Job为eng实例注册了4个Handler，Handler名分别为：allocate_interface、release_interface、allocate_port和link。

Docker Daemon的网络初始化关乎Docker容器的通信能力，是Docker架构中最为基础的知识之一。第6章将为大家分析Docker Daemon网络环境的创建。

4.7.1　创建Docker网络设备

创建Docker网络设备，属于Docker Daemon创建网络环境的第一步，实际工作是创建名为docker0的网桥设备。

在InitDriver函数运行过程中，Docker首先使用Job的环境变量初始化内部变量；然后根据目前网络环境，判断是否创建docker0网桥，若Docker专属网桥已存在，则继续往下执行；否则，创建docker0网桥。具体实现为createBridge（bridgeIP），以及createBridgeIface（bridgeIface）。

createBridge的功能是：在宿主机上启动创建指定名称网桥设备的任务，并为该网桥设备配置一个与其他设备不冲突的网络地址。而createBridgeIface通过系统调用负责创建具体实际的网桥设备，并设置MAC地址，通过libcontainer中netlink包的CreateBridge来实现。

4.7.2　启用iptables功能

创建完网桥之后，Docker Daemon为未来的Docker容器以及宿主机配置iptables规则，作用是：为Docker容器之间的link操作提供iptables防火墙支持。源码位于./docker/docker/daemon/networkdriver/bridge/driver/driver.go#L133-L137，如下所示：

```
// Configure iptables for link support
if enableIPTables {
     if err := setupIPTables(addr, icc); err != nil {
          return job.Error(err)
     }
}
```

其中setupIPtables的调用过程中，addr地址为Docker网桥的网络地址，icc为true，即允许Docker容器间互相访问。假设网桥设备名为docker0，网桥网络地址为docker0_ip，设置iptables规则，具体操作步骤如下：

1）使用iptables工具开启新建网桥的NAT功能，使用如下命令：

```
iptables -I POSTROUTING -t nat -s docker0_ip ! -o docker0 -j MASQUERADE
```

2） 通过icc参数，决定是否允许Docker容器间的通信，并制定相应iptables的Forward链。Docker容器之间建立通信，说明数据包从源容器内发出后，经过docker0，并且还需要在docker0处发往docker0，最终转向目 标容器。换言之，从docker0出来的数据包，如果需要继续发往docker0，则说明是Docker容器间的通信数据包。使用命令如下：

```
iptables -I FORWARD -i docker0 -o docker0 -j ACCEPT
```

3） 允许接受从容器发出，且目 标地址不是容器的数据包。换言之，允许所有从docker0发出且不是继续发向docker0的数据包，使用命令如下：

```
iptables -I FORWARD -i docker0 ! -o docker0 -j ACCEPT
```

4） 对于发往docker0，并且属于已经建立的连接的数据包，Docker无条件接受这些连接上的数据包，使用命令如下：

```
iptables -I FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j
ACCEPT
```

4.7.3 启用系统数据包转发功能

在Linux系统上，网络设备之间的数据包转发功能是默认禁止的。数据包转发，即当宿主机存在多块网络设备时，如果其中一块网络设备接收到数据包，则无条件将其转发给另外的网络设备。通过修改/proc/sys/net/ipv4/ip_forward的值，将其置为1，则可以保证系统内数据包可以实现转发功能，代码如下：

```
if ipForward {

     // Enable IPv4 forwarding

          if err := ioutil.WriteFile("/proc/sys/net/ipv4/ip_forward", []byte{'1',
'\n'}, 0644); err != nil {

          job.Logf("WARNING: unable to enable IPv4 forwarding: %s\n", err)

     }

}
```

4.7.4 创建DOCKER链

Docker在网桥设备上创建一条名为DOCKER的链，该链的作用是在创建Docker容器时实现容器与宿主机的端口映射。实现代码位于./docker/docker/daemon/networkdriver/bridge/driver/driver.go，如下所示：

```
if err := iptables.RemoveExistingChain("DOCKER"); err != nil {
    return job.Error(err) }
if enableIPTables {
    chain, err := iptables.NewChain("DOCKER", bridgeIface) if err != nil {
        return job.Error(err) }
      portmapper.SetIptablesChain(chain) }
```

4.7.5　注册处理方法至Engine

创建完网桥，并配置完基本的iptables规则之后，Docker Daemon在网络方面还向Engine中注册了4个处理方法，这些处理方法的名称与作用如下：

·allocate_interface：为Docker容器分配专属网络接口，分配容器网段的IP地址；

·realease_interface：释放Docker容器占用的网络接口资源；

·allocate_port：为Docker容器分配一个端口；

·link：实现Docker容器间的连接操作。

4.8 创建graphdb并初始化

graphdb是一个构建在SQLite之上的图形数据库，通常用来记录节点命名以及节点之间的关联。在Docker的世界中，用户可以通过link操作，使得Docker容器之间建立一种关联，而Docker Daemon正是使用graphdb来记录这种容器间的关联信息。Docker创建graphdb的源码如下：

```
graphdbPath := path.Join(config.Root, "linkgraph.db")

graph, err := graphdb.NewSqliteConn(graphdbPath)

if err != nil {

     return nil, err

}
```

以上代码首先确定graphdb的目录为/var/lib/docker/linkgraph.db；随后通过graphdb包内的NewSqliteConn打开graphdb，使用的驱动为sqlite3，数据源的名称为/var/lib/docker/linkgraph.db；最后通过NewDatabase函数初始化整个graphdb，为graphdb创建entity表和edge表，并在这两个表中初始化部分数据。NewSqliteConn函数的实现位

于./docker/docker/pkg/graphdb/conn_sqlite3.go，代码实现如下：

```
func NewSqliteConn(root string) (*Database, error) {

    ...

    conn, err := sql.Open("sqlite3", root)

    ...

    return NewDatabase(conn, initDatabase)

}
```

4.9 创建execdriver

execdriver是Docker中用来执行Docker容器任务的驱动。创建并初始化graphdb之后，Docker Daemon随即开始创建了execdriver，具体源码如下：

```
ed, err := execdrivers.NewDriver(config.ExecDriver, config.Root, sysInitPath,
sysInfo)
```

分析以上源码可知，DockerDaemon创建execdriver时，需要以下四部分信息。

·config.ExecDriver：Docker运行时中用户指定使用的execdriver类型，在默认配置文件中值为native。用户也可以在启动DockerDaemon将这个值配置为lxc，则导致Docker使用lxc类型的驱动执行Docker容器的内部操作。

·config.Root：Docker运行时的root路径，默认配置文件中为/var/lib/docker。

·sysInitPath：系统中存放dockerinit二进制文件的路径，一般为/var/lib/docker/init/dockerinit-1.2.0。

·sysInfo：系统功能信息，包括：容器的内存限制功能，交换区内存限制功能，数据转发功能，以及AppArmor安全功能等。

在执行execdrivers.NewDriver之前，首先通过以下代码，获取期望的目 标dockerinit文件的路径localPath，以及系统中dockerinit文件实际所在的路径sysInitPath：

```
localCopy := path.Join(config.Root, "init", fmt.Sprintf("dockerinit-%s",
dockerversion.VERSION))
sysInitPath := utils.DockerInitPath(localCopy)
```

通过执行以上代码，localCopy为/var/lib/docker/init/dockerinit-1.2.0，而sysyInitPath为当前Docker运行时中dockerinit-1.2.0实际所处的路径，utils.DockerInitPath实现位于./docker/docker/utils/util.go。若localCopy与sysyInitPath不相等，则说明当前系统中的dockerinit二进制文件，不在localCopy路径下，则需要将其复制至localCopy下，并对该文件设定权限。

设定完dockerinit二进制文件的位置之后，Docker Daemon创建sysinfo对象，记录系统的功能属性。SysInfo的定义位于./docker/docker/pkg/sysinfo/sysinfo.go，如下所示：

```
type SysInfo struct {
    MemoryLimit               bool
    SwapLimit                 bool
    IPv4ForwardingDisabled    bool
```

```
    AppArmor                    bool
}
```

其中Docker Daemon通过判断cgroups文件系统挂载路径下是否均存在memory.limit_in_bytes和memory.soft_limit_in_bytes文件来为MemoryLimit赋值，若均存在，则置为true，否则置为false；通过判断memory.memsw.limit_in_bytes文件是否存在来为SwapLimit赋值，若该文件存在，则置为true，否则置为false。AppArmor的值则是通过宿主机上是否存在/sys/kernel/security/apparmor来判断，若存在，则置为true，否则置为false。

执行execdrivers.NewDriver时，返回execdriver.Driver对象实例，具体代码实现位于./docker/docker/daemon/execdriver/execdrivers/execdrivers.go，由于选择使用native作为exec驱动，故执行以下代码，返回最终的execdriver，其中native.NewDriver实现位于./docker/docker/daemon/execdriver/native/driver.go：

```
return native.NewDriver(path.Join(root, "execdriver", "native"), initPath)
```

至此，一个execdriver的实例ed被Docker成功创建。

4.10 创建daemon实例

Docker Daemon在经过以上多个环节的设置之后，整合众多已经创建的对象，创建最终的Daemon对象实例daemon。Daemon对象实例daemon涉及的内容极多，比如：对于Docker镜像的存储可以通过graph来管理、所有Docker容器的元数据信息都保存在containers对象中、整个Docker Daemon的任务执行位于eng属性中，等等。

创建daemon实例的源码实现如下：

```
daemon := &Daemon{
     repository:     daemonRepo,
     containers:     &contStore{s: make(map[string]*Container)},
     graph:          g,
     repositories:   repositories,
     idIndex:        truncindex.NewTruncIndex([]string{}),
     sysInfo:        sysInfo,
     volumes:        volumes,
     config:          config,
     containerGraph: graph,
     driver:         driver,
     sysInitPath:    sysInitPath,
     execDriver:     ed,
     eng:            eng,
}
```

分析Daemon类型的属性如表4-1所示。

表4-1　Daemon类型属性分析表

属性名	作用
repository	存储所有 Docker 容器信息的路径，默认为 /var/lib/docker/containers
containers	用于存储 Docker 容器信息的对象

（续）

属性名	作用
graph	存储 Docker 镜像的 graph 对象
repositories	存储本机所有 Docker 镜像 repo 信息的对象
idIndex	用于通过简短有效的字符串前缀定位唯一的镜像
sysInfo	系统功能信息
volumes	管理宿主机上 volumes 内容的 graphdriver，默认为 vfs 类型
config	Config.go 文件中的配置信息，以及执行后产生的配置 DisableNetwork
containerGraph	存放 Docker 镜像关系的 graphdb
driver	管理 Docker 镜像的驱动 graphdriver，默认为 aufs 类型
sysInitPath	系统 dockerinit 二进制文件所在的路径
execDriver	Docker Daemon 的 exec 驱动，默认为 native 类型
eng	Docker 的执行引擎 Engine 类型

4.11 检测DNS配置

创建完Daemon类型实例daemon之后，Docker Daemon使用daemon.checkLocaldns（ ）检测Docker运行环境中DNS的配置，checkLocaldns函数的定义位于./docker/docker/daemon/daemon.go#L854-L856，代码如下：

```
func (daemon *Daemon) checkLocaldns() error {
     resolvConf, err := resolvconf.Get()
     if err != nil {
          return err
     }
     if len(daemon.config.Dns) == 0 && utils.CheckLocalDns(resolvConf) {
               log.Infof("Local (127.0.0.1) DNS resolver found in resolv.conf and containers can't use it. Using default external servers : %v", DefaultDns)
          daemon.config.Dns = DefaultDns
     }
     return nil
```

```
}
```

以上代码首先通过resolvconf.Get（ ） 方法获取宿主机/etc/resolv.conf中的DNS服务器信息。若宿主机上DNS文件resolv.conf中有127.0.0.1，而Docker容器在自身内部不能使用该地址，故采用默认外在DNS服务器，为8.8.8.8，8.8.4.4；若宿主机上的resolv.conf有Docker容器可以使用的DNS服务器地址，则Docker Daemon采用该地址。最终Docker Daemon将DNS服务器地址赋值给config文件中的Dns属性。用户通过Docker Daemon创建Docker容器时，若不指定DNS服务器地址，则Docker Daemon将会使用daemon.Config.Dns作为容器内部的DNS服务器地址。

4.12　启动时加载已有Docker容器

在Docker Daemon重启时，很有可能之前有遗留的Docker容器。对于这部分容器，DockerDaemon重启前，一直将元数据信息存储在daemon.repository（目录为/var/lib/docker/containers）中。为了保证重启之后Docker容器的信息不丢失，DockerDaemon首先会进入该目录去查看，是否存在遗留的Docker容器。若存在，则Docker Daemon加载这部分容器，将容器信息收集，并做相应的维护。

Docker Daemon加载Docker容器的源码实现位于./docker/docker/daemon/daemon.go#L854-L856，如下：

```
if err := daemon.restore(); err != nil {
	return nil, err
}
```

需要注意的是：由于Docker Daemon的重启不会重启所有重启前运行的容器，故Docker Daemon加载已有容器时，会判断容器之前的状态是否为运行，若是的话，会将该容器的状态置为退出，并在内存中的容器对象以及config.json文件中将容器主进程的PID设为0。

4.13 设置shutdown的处理方法

加载完已有Docker容器之后，Docker Daemon设置了多项在shutdown操作中需要执行的处理方法。也就是说，当Docker Daemon接收到特定信号，需要执行shutdown操作时，先执行这些处理方法完成善后工作，最终再实现物理意义上的shutdown。实现源码如下：

```
eng.OnShutdown(func() {
     if err := daemon.shutdown(); err != nil {
          log.Errorf("daemon.shutdown(): %s", err)
     }
     if err := portallocator.ReleaseAll(); err != nil {
          log.Errorf("portallocator.ReleaseAll(): %s", err)
     }
     if err := daemon.driver.Cleanup(); err != nil {
          log.Errorf("daemon.driver.Cleanup(): %s", err.Error())
     }
     if err := daemon.containerGraph.Close(); err != nil {
          log.Errorf("daemon.containerGraph.Close(): %s", err.Error())
     }
})
```

由以上源码可见，eng对象shutdown操作执行时，需要执行以上作为参数的func（ ）{……}函数。在该函数中，主要完成以下4部分操作：

·运行daemon对象的shutdown函数，做daemon方面的善后工作。

·通过portallocator.ReleaseAll（），释放所有之前占用的端口资源。

·通过daemon.driver.Cleanup（），通过graphdriver实现unmount所有有关镜像layer的挂载点。

·通过daemon.containerGraph.Close（）关闭graphdb的连接。

4.14 返回daemon对象实例

当所有的工作完成之后，Docker Daemon返回daemon实例，意味着NewDaemon函数执行完毕，程序运行最终返回至mainDaemon（）中，继续通过goroutine完成加载daemon。

4.15 总结

本章从源码的角度深度分析了Docker Daemon启动过程中daemon对象的创建与加载。在这一环节中涉及内容极多，本章着重归纳总结daemon实现的逻辑，一一深入，具体全面。

在Docker的架构中，Docker Daemon的内容是最为丰富全面的，而NewDaemon的实现涵盖了Docker Daemon启动过程中的大部分工作。可以认为NewDaemon是Docker Daemon实现过程中的核心所在。深入理解NewDaemon的实现，即掌握了Docker Daemon运行的来龙去脉。

第5章　Docker Server的创建

5.1　引言

Docker架构中，Docker Server是Docker Daemon的重要组成部分。Docker Server最主要的功能是：接收用户通过Docker Client发送的请求，并按照相应的路由规则实现请求的路由分发，最终将请求处理后的结果返回至Docker Client。

同时，Docker Server具备十分优秀的用户友好性，多种通信协议的支持大大降低Docker用户使用Docker的门槛。除此之外，Docker Server设计实现了详尽清晰的API接口，以供Docker用户使用。通信安全方面，Docker Server可以提供安全传输层协议（TLS），保证Docker Client与Docker Server之间数据的加密传输。并发处理方面，Docker Daemon大量使用了Go语言中的协程goroutine，大大提高了服务端对于请求的并发处理能力。

本章将从源码的角度分析Docker Server的创建，分析内容的安排如下：

1）介绍Job“serveapi”的创建与执行流程，代表Docker Server的创建。

2） 深入分析Job“serveapi”的执行流程。

3） 分析Docker Server创建Listener并服务API的流程。

5.2 Docker Server创建流程

我们在第3章主要分析了Docker Daemon的启动，而在mainDaemon（）运行的最后环节，Docker实现了创建并运行名为serveapi的Job。这一环节的作用是：让Docker Daemon提供API访问服务。实质上，这正是实现了Docker架构中Docker Server的创建与运行。

从流程的角度来说，Docker Server的创建与运行，代表了Job“serveapi”的整个生命周期。整个生命周期包括：创建Docker Server的Job，配置Job环境变量，以及触发执行Job。

5.2.1　创建名为“serveapi”的Job

Docker架构中，Job是Engine内部最基本的任务执行单位。创建Docker Server，服务于API请求，同样属于Docker内部的一项任务。因此，这一任务同样需要表示为一个可执行的Job。换言之，需要创建Docker Server，则必须创建一个相应的Job。具体的Job创建形式位于./docker/docker/docker/daemon.go#L66，如下所示：

```
job := eng.Job("serveapi", flHosts...)
```

以上代码通过Engine实例eng创建一个Job类型的实例job，Job实例名为serveapi，同时用flHosts的值初始化job.Args。flHosts的作用是：配置Docker Server监听的协议与监听的地址。

需要注意的是，第3章中函数mainDaemon（ ）的具体实现过程中，在加载builtins环节已经向eng对象注册了键为serveapi的处理方法，而该处理方法的值为api.ServeApi。因此，在运行名为serveapi的Job时，会执行该Job的处理方法api.ServeApi。

5.2.2　配置Job环境变量

创建完Job实例job之后，Docker Daemon为Job实例配置环境参数。在Job的实现过程中，为Job配置参数的方式有两种：第一种，创建Job实例时，用指定参数直接初始化Job的Args属性；第二种，创建完Job后，给Job添加指定的环境变量。以下源码实现了为创建的job配置环境变量。

```
job.SetenvBool("Logging", true)
job.SetenvBool("EnableCors", *flEnableCors)
job.Setenv("Version", dockerversion.VERSION)
job.Setenv("SocketGroup", *flSocketGroup)
job.SetenvBool("Tls", *flTls)
job.SetenvBool("TlsVerify", *flTlsVerify)
job.Setenv("TlsCa", *flCa)
job.Setenv("TlsCert", *flCert)
job.Setenv("TlsKey", *flKey)
job.SetenvBool("BufferRequests", true)
```

对于以上配置环境变量的归纳总结如表5-1所示。

表5-1　Job环境变量列表

环境变量名	flag 参数	默认值	作用值
Logging		true	启用 Docker 容器的日志输出
EnableCors	flEnableCors	false	在远程 API 中提供 CORS 头
Version			显示 Docker 版本号
SocketGroup	flSocketGroup	docker	在 daemon 模式中 unix domain socket 分配用户组名
Tls	flTls	false	使用 TLS 安全传输协议
TlsVerify	flTlsVerify	false	使用 TLS 并验证远程客户端
TlsCa	flCa		指定 CA 文件路径
TlsCert	flCert		TLS 证书文件路径
TlsKey	flKey		TLS 密钥文件路径
BufferRequest		true	缓存 Docker Client 请求

5.2.3　运行Job

创建完Job，配置完Job的环境变量，意味着DockerServer的创建需求已准备完毕。万事俱备，只欠东风，东风就是触发执行这个Job。Docker中通过Job实例的run函数完成Job的触发执行。触发执行serveapi这个Job的具体实现源码如下：

```
if err := job.Run(); err != nil {
        log.Fatal(err)
}
```

由于Docker已经在eng对象中注册过键为serveapi的处理方法，故在运行job的时候，执行这个处理方法的值函数，相应处理方法的值为api.ServeApi。至此，名为serveapi的Job的生命周期已经完备。本章余下内容将深入分析Job的处理方法，api.ServeApi的执行细节。

5.3　ServeApi运行流程

ServeApi属于Docker Server提供API服务的部分，本小节将从源码的角度剖析Docker Server的架构设计与实现。

作为一个监听请求、处理请求、响应请求的服务端，Docker Server首先需要明确自身可以为多少种通信协议提供服务。稍加深入学习Docker这个C/S模式的架构设计，就可以发现Docker Server支持的协议包括以下三种：TCP协议、UNIX Socket形式以及fd的形式。随后，Docker Server根据协议的不同，分别创建不同的服务端实例。最后，在不同的服务端实例中，创建相应的路由模块、监听模块，以及处理请求的处理方法，形成一个完备的服务端。

serveapi这个Job在运行时，将执行api.ServeApi函数。ServeApi的功能是：循环检查Docker Daemon当前支持的所有通信协议，并为每一种协议都创建一个协程goroutine，并在此协程内部配置一个服务于HTTP请求的服务端。ServeApi的源码实现位于./docker/docker/api/server/server.go#L1339，如下所示：

```
func ServeApi(job *engine.Job) engine.Status {
	if len(job.Args) == 0 {
		return job.Errorf("usage: %s PROTO://ADDR [PROTO://ADDR ...]",
job.Name)
	}
	var (
		protoAddrs = job.Args
		chErrors   = make(chan error, len(protoAddrs))
```

```
	)
	activationLock = make(chan struct{})
	for _, protoAddr := range protoAddrs {
		protoAddrParts := strings.SplitN(protoAddr, "://", 2)
		if len(protoAddrParts) != 2 {
			return job.Errorf("usage: %s PROTO://ADDR [PROTO://ADDR ...]",
job.Name)
		}
		go func() {
				log.Infof("Listening for HTTP on %s (%s)", protoAddrParts[0],
protoAddrParts[1])
				chErrors <- ListenAndServe(protoAddrParts[0],
protoAddrParts[1], job)
		}()
	}
	for i := 0; i < len(protoAddrs); i += 1 {
		err := <-chErrors
		if err != nil {
			return job.Error(err)
		}
	}
	return engine.StatusOK
}
```

分析以上源码，通过模块化的划分，我们可以发现ServeApi的执行流程主要分为以下4个步骤：

1） 检验Job的参数，确保传入参数无误。

2） 定义Docker Server的监听协议与地址，以及错误信息管道channel。

3） 遍历协议地址，针对协议创建相应的服务端。

4） 通过chErrors建立goroutine与主进程之间的协调关系。

下面详细分析以上4个步骤：

第一，Docker Daemon判断Job的参数，保证传入的Job参数无误。DockerDaemon判断的依据来源于job.Args的长度。由于Docker创建serveapi这个Job时，通过flHosts来初始化job.Args，故job.Args为相当于数组flHost，若flHost的长度为0，则说明Docker Server没有监听的协议与地址，参数有误，返回错误信息。源码如下：

```
if len(job.Args) == 0 {
    return job.Errorf("usage: %s PROTO://ADDR [PROTO://ADDR ...]", job.Name)
}
```

第二，定义protoAddrs、chErrors与activationLock三个变量，分别代表Docker Server监听的协议与地址，以及Job间的同步channel。

protoAddrs代表flHosts的内容；而chError定义了和protoAddrs长度一致的错误类型管道，chError的作用会在下文中说明。同时定义的变量activationLock，是用以同步serveapi和acceptconnections这两个Job执行的管道。serveapi运行时，ServeFd和ListenAndServe函数均由于activationLock中没有内容而阻塞，而当运行acceptionconnections这个Job时，该Job会首先通知init进程Docker Daemon已经启动完毕，并关闭activationLock，因此ServeFd以及ListenAndServe不再阻塞，结果是serveapi继续执行。正是由于activationLock的存在，Docker

Daemon可以保证acceptconnections这个Job的运行有能力通知serveapi开启正式服务于API请求的功能。源码如下：

```
var (
     protoAddrs = job.Args
     chErrors   = make(chan error, len(protoAddrs))
)
activationLock = make(chan struct{})
```

第三，遍历协议地址，针对协议创建相应的服务端。协议地址即protoAddrs，也就是job.Args。DockerDaemon将protoAddrs的每一元素都按照字符串“：//”进行分割，若分割后protoAddrParts的长度不为2，则说明协议地址的书写形式有误，返回Job错误；若分割后protoAddrParts的长度为2，则说明地址协议符合标准，获取protoAddrParts中的协议protoAddrParts[0]与地址protoAddrParts[1]。最后，针对每一次循环中获得的协议与地址，Docker Daemon均创建一个goroutine来执行ListenAndServe的操作。goroutine的运行主要依赖于ListenAndServe（protoAddrParts[0]，protoAddrParts[1]，job）的运行结果。若ListenAndServe返回错误，则chErrors中有错误，当前协程执行完毕；若没有返回错误，则该协程持续运行，持续提供服务。其中最为重要的是ListenAndServe的实现，该函数具体实现了Docker Daemon如何创建listener、router以及server，并协调三者进行工作，最终服务于API请求。步骤三的源码实现如下：

```
for _, protoAddr := range protoAddrs {
     protoAddrParts := strings.SplitN(protoAddr, "://", 2)
     if len(protoAddrParts) != 2 {
          return job.Errorf("usage: %s PROTO://ADDR [PROTO://ADDR ...]",
job.Name)
     }
     go func() {
          log.Infof("Listening for HTTP on %s (%s)", protoAddrParts[0],
protoAddrParts[1])
          chErrors <- ListenAndServe(protoAddrParts[0], protoAddrParts[1], job)
     }()
}
```

第四，根据chErrors的值运行，若chErrors这个管道中有错误内容，则ServeApi的一次循环结束；若无错误内容，则循环被阻塞。chErrors这个管道的作用是：确保ListenAndServe所对应的协程能和主函数ServeApi进行协调，如果协程运行出错，主函数ServeApi仍然可以捕获这样的错误，从而导致程序的退出。实现源码如下：

```
for i := 0; i < len(protoAddrs); i += 1 {
     err := <-chErrors
     if err != nil {
          return job.Error(err)
     }
}
return engine.StatusOK
```

至此，ServeApi的运行流程已经全部分析完毕，其中核心部分ListenAndServe的实现，将在5.4节深入分析。

5.4 ListenAndServe实现

ListenAndServe的功能是：使Docker Server监听某一指定地址，并接收该地址上的请求，并对以上请求路由转发至相应的处理方法处。从实现的角度来看，ListenAndServe主要实现了设置一个服务于HTTP协议请求的服务端，该服务端监听指定地址上的请求，并对请求做特定的协议检查，最终完成请求的路由与分发。代码实现位于./docker/docker/api/server/server.go。

ListenAndServe的实现可以分为以下4个部分：

1）创建router路由实例。

2）创建listener监听实例。

3）创建http.Server。

4）启动API服务。

ListenAndServe的执行流程如图5-1所示。

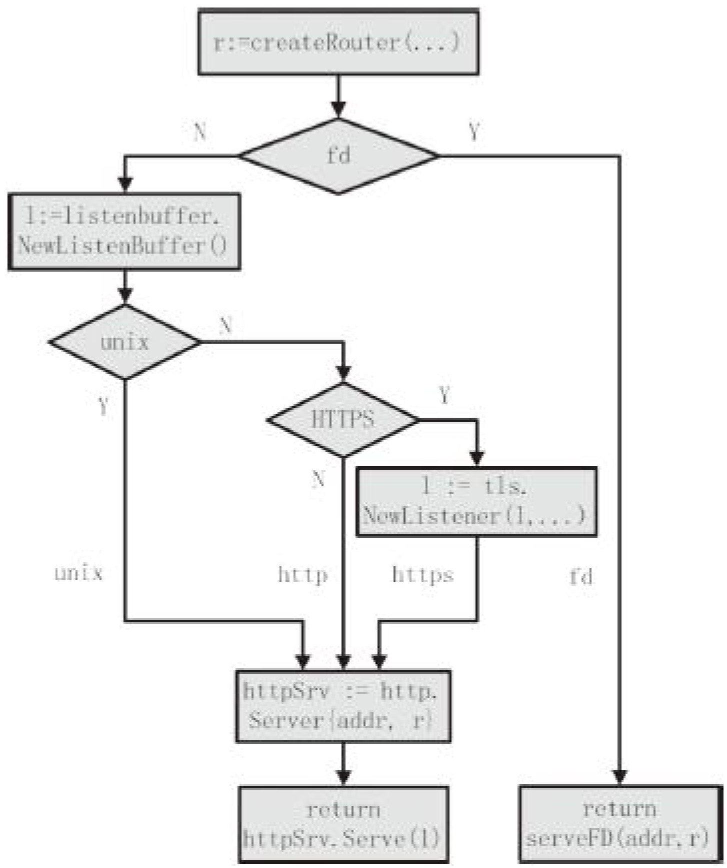


图5-1　ListenAndServe执行流程图

本节将按照ListenAndServe执行流程图一一深入分析各个部分。

5.4.1　创建router路由实例

首先，在函数ListenAndServe的实现过程中，Docker通过createRouter创建了一个router路由实例。代码源码如下：

```
r, err := createRouter(job.Eng, job.GetenvBool("Logging"),
job.GetenvBool("EnableCors"), job.Getenv("Version"))
if err != nil {
     return err
}
```

createRouter的实现位于./docker/docker/api/server/server.go#L1094。

创建router路由实例是一个重要的环节，路由实例的作用是：负责Docker Server对外部请求的路由以及分发。在实现过程中，有两个主要步骤：第一，创建全新的router路由实例；第二，为router实例添加路由记录。

1.创建空路由实例

实质上，createRouter通过包gorilla/mux来实现一个功能强大的路由器和分发器。源码实现如下：

```
r := mux.NewRouter()
```

NewRouter（ ）函数返回了一个全新的router实例r。在创建Router实例时，给Router对象实例的两个属性进行赋值，分别为nameRoutes和KeepContext。其中namedRoutes属性为map类型，键为string类型，值为Route路由记录类型；另外，KeepContext属性为false，表示Docker Server在处理完请求之后，就清除请求的内容，不对请求做存储操作。源码位于./docker/docker/vendor/src/github.com/gorilla/mux/mux.go#L16，如下所示：

```
func NewRouter() *Router {
     return &Router{namedRoutes: make(map[string]*Route), KeepContext: false}
}
```

可见，以上代码返回的类型为mux.Router。mux.Router会通过一系列已经注册过的路由记录，来匹配接收的请求。首先通过请求的URL或者其他条件，找到相应的路由记录，并调用这条路由记录中的执行处理方法。mux.Router有以下特性：

·请求可以基于URL的主机名、路径、路径前缀、shemes、请求头和请求值、HTTP请求方法类型或者使用自定义的匹配规则。

·URL主机名和路径可以通过一个正则表达式来表示。

·注册的URL可以被直接运用，也可以被保留，从而保证维护资源的使用。

·路由记录同样可以作用于子路由记录：如果父路由记录匹配，则嵌套记录只会被用来测试。当设计一个组内的路由记录共享相同的匹配条件时，如主机名、路径前缀或者其他重复的属性，子路由的方式会起到相应的效果。

·mux.Router实现了http.Handler接口，故和标准的http.ServeMux兼容。

2.添加路由记录

Router路由实例r创建完毕，下一步工作是为Router实例r添加所需要的路由记录。路由记录存储着用来匹配请求的信息，包括对请求的匹配规则，以及匹配之后的处理方法执行入口。

回到createRouter实现源码中，Docker首先判断Docker Daemon的启动过程中有没有开启DEBUG模式。通过docker可执行文件启动Docker Daemon，解析flag参数时，若flDebug的值为false，则说明不需要配置DEBUG环境；若flDebug的值为true，则说明需要为Docker Daemon添加DEBUG功能。具体的源码实现如下：

```
if os.Getenv("DEBUG") != "" {
      AttachProfiler(r)
}
```

AttachProiler（r）的功能是为路由实例r添加与DEBUG相关的路由记录，具体实现位于./docker/docker/api/server/server.go#L1083，如下所示：

```
func AttachProfiler(router *mux.Router) {
     router.HandleFunc("/debug/vars", expvarHandler)
     router.HandleFunc("/debug/pprof/", pprof.Index)
     router.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
     router.HandleFunc("/debug/pprof/profile", pprof.Profile)
     router.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
     router.HandleFunc("/debug/pprof/heap", pprof.Handler("heap").ServeHTTP)
     router.HandleFunc("/debug/pprof/goroutine",
pprof.Handler("goroutine").ServeHTTP)
          router.HandleFunc("/debug/pprof/threadcreate", pprof.Handler
("threadcreate").ServeHTTP)
}
```

分析以上源码，可以发现Docker Server使用两个包来完成DEBUG相关的工作：expvar和pprof。包expvar为公有变量提供标准化的接口，使得这些公有变量可以通过HTTP的形式在“/debug/vars”这个URL下被访问，传输格式为JSON。包pprof将Docker Server运行时的分析数据通过“/debug/pprof/”这个URL向外暴露。这些运行时信息包括以下内容：可得的信息列表、正在运行的命令行信息、CPU信息、程序函数引用信息、ServeHTTP函数三部分信息的使用情况（堆使用、协程使用和线程使用）。

再次回到createRouter函数实现中，完成DEBUG功能的所有工作之后，Docker创建了一个映射类型的对象m，用于初始化路由实例r的路由记录。简化的m对象，源码如下：

```
m := map[string]map[string]HttpApiFunc{
    "GET": {
        "/events":                          getEvents,
        "/info":                            getInfo,
        "/version":                         getVersion,
        "/images/json":                     getImagesJSON,
        "/images/viz":                      getImagesViz,
        "/images/search":                   getImagesSearch,
        "/images/{name:.*}/get":            getImagesGet,
        "/images/{name:.*}/history":        getImagesHistory,
        "/images/{name:.*}/json":           getImagesByName,
        "/containers/ps":                   getContainersJSON,
        "/containers/json":                 getContainersJSON,
        ...
    },
    "POST": {
        ...
        "/containers/{name:.*}/copy":    postContainersCopy,
    },
    "DELETE": {
        "/containers/{name:.*}": deleteContainers,
        "/images/{name:.*}":     deleteImages,
    },
    "OPTIONS": {
        "": optionsHandler,
    },
}
```

对象m的类型为映射，其中键为string类型，代表HTTP的请求类型，如GET、POST、DELETE等，值为另一个映射类型，该映射代表的是URL与执行处理方法的映射。在第二个映射类型中，键为string类型，代表的是请求URL，值为HttpApiFunc类型，代表具体的执行处理方法。其中HttpApiFunc类型的定义如下：

```
type HttpApiFunc func(eng *engine.Engine, version version.Version, w
http.ResponseWriter, r *http.Request, vars map[string]string) error
```

完成对象m的定义，随后Docker Server通过该对象m来添加路由实例r的路由记录。对象m的请求方法、请求URL和请求处理方法这三样内容可以为对象r构建一条路由记录。源码实现如下：

```
for method, routes := range m {
     for route, fct := range routes {
          log.Debugf("Registering %s, %s", method, route)
          localRoute := route
          localFct := fct
          localMethod := method
               f := makeHttpHandler(eng, logging, localMethod, localRoute,
localFct, enableCors, version.Version(dockerVersion))
          if localRoute == "" {
               r.Methods(localMethod).HandlerFunc(f)
          } else {
                    r.Path("/v{version:[0-9.]+}"
localRoute).Methods(localMethod).HandlerFunc(f)
                    r.Path(localRoute).Methods(localMethod).HandlerFunc(f)
          }
     }
}
```

分析以上源码可以发现：在第一层循环中，按HTTP请求方法划分，获得请求方法各自的路由记录；第二层循环，按匹配请求的URL进行划分，获得与URL相对应的执行处理方法。在嵌套循环中，通过makeHttpHandler返回一个执行的函数f。在返回的这个函数中，涉及了日志配置信息、CORS信息（跨域资源共享协议），以及版本信息。下面举例说明makeHttpHandler的实现，从对象m可以看到，对于GET请求，请求URL为/info，则请求处理方法为getInfo。执行makeHttpHandler的具体源码实现如下：

```
func makeHttpHandler(eng *engine.Engine, logging bool, localMethod string,
localRoute string, handlerFunc HttpApiFunc, enableCors bool, dockerVersion
version.Version) http.HandlerFunc {
     return func(w http.ResponseWriter, r *http.Request) {
          // log the request
          log.Debugf("Calling %s %s", localMethod, localRoute)
          if logging {
               log.Infof("%s %s", r.Method, r.RequestURI)
          }
          if strings.Contains(r.Header.Get("User-Agent"), "Docker-Client/") {
               userAgent := strings.Split(r.Header.Get("User-Agent"), "/")
                    if len(userAgent) == 2 &&
!dockerVersion.Equal(version.Version(userAgent[1])) {
                    log.Debugf("Warning: client and server don't have the same
version (client: %s, server: %s)", userAgent[1], dockerVersion)
               }
```

```
        }
        version := version.Version(mux.Vars(r)["version"])
        if version == "" {
            version = api.APIVERSION
        }
        if enableCors {
            writeCorsHeaders(w, r)
        }
        if version.GreaterThan(api.APIVERSION) {
                http.Error(w, fmt.Errorf("client and server don't have same
version (client : %s, server: %s)", version, api.APIVERSION).Error(),
http.StatusNotFound)
            return
        }
        if err := handlerFunc(eng, version, w, r, mux.Vars(r)); err != nil {
                log.Errorf("Handler for %s %s returned error: %s",
localMethod, localRoute, err)
            httpError(w, err)
        }
    }
}
```

可见makeHttpHandler的执行直接返回一个函数func(w http.ResponseWriter，r*http.Request) 。在func函数的实现中，判断makeHttpHandler传入的logging参数，若为true，则将该处理方法的执行日志显示出来；另外通过makeHttpHandler传入的enableCors参数判断是否在HTTP请求的头文件中添加跨域资源共享信息，若为true，则通过writeCorsHeaders函数向请求响应中添加有关CORS的HTTP Header，源码实现位于./docker/docker/api/server/server.go#L1022，如下所示：

```
func writeCorsHeaders(w http.ResponseWriter, r *http.Request) {
    w.Header().Add("Access-Control-Allow-Origin", "*")
        w.Header().Add("Access-Control-Allow-Headers", "Origin, X-Requested-
With, Content-Type, Accept")
        w.Header().Add("Access-Control-Allow-Methods", "GET, POST, DELETE, PUT,
OPTIONS")
}
```

最为重要的执行部分为handlerFunc（eng，version，w，r，mux.Vars（r）） ，源码如下：

```
if err := handlerFunc(eng, version, w, r, mux.Vars(r)); err != nil {
       log.Errorf("Handler for %s %s returned error: %s", localMethod,
       localRoute, err)
       httpError(w, err)
}
```

对于GET请求类型，URL为/info的请求，由于处理方法名为getInfo，也就是说handlerFunc这个行参的值为getInfo，故执行部分直接运行getInfo（eng，version，w，r，mux.Vars（r）） ，而getInfo的具体实现位于./docker/docker/api/server/serve.go#L269，如下所示：

```
func getInfo(eng *engine.Engine, version version.Version, w http.ResponseWriter,
r *http.Request, vars map[string]string) error {
    w.Header().Set("Content-Type", "application/json")
    eng.ServeHTTP(w, r)
    return nil
}
```

以上makeHttpHandler的执行已经完毕，返回func函数，作为指定URL对应的处理方法。

创建完处理函数后，Docker需要向路由实例添加新的路由记录。如果URL信息为空，则直接为该HTTP请求方法类型添加路由记录；若URL不为空，则为请求URL路径添加新的路由记录。需要额外注意的是，在URL不为空，为路由实例r添加路由记录时，考虑了API版本

的问题，通过r.Path（"/v{version：[0-9.]+}"+localRoute）.Methods（localMethod）.HandlerFunc（f）来实现。

至此，mux.Router实例r的两部分工作已经全部完成：创建空的路由实例r，为r添加相应的路由记录，最后返回配置后的路由实例r。

5.4.2　创建listener监听实例

路由模块，完成请求的路由与分发，是ListenAndServe实现中的第一项重要工作。对于请求的监听功能，同样需要模块来完成。而在ListenAndServe实现中，第二项重要的工作就是创建Listener。Listener是一种面向流协议的通用网络监听模块。

在创建Listener之前，Docker先判断Docker Server允许的协议，若协议为fd形式，则直接通过ServeFd来服务请求；若协议不为fd形式，则继续往下执行。

程序首先需要判断serveapi这个Job的环境中BufferRequests的值，若为true，则通过包listenbuffer创建一个Listener的实例l，否则的话直接通过包net创建Listener实例l。具体的源码位于./docker/docker/api/server/server.go#L1269-L1273，如下所示：

```
if job.GetenvBool("BufferRequests") {
     l, err = listenbuffer.NewListenBuffer(proto, addr, activationLock)
} else {
     l, err = net.Listen(proto, addr)
}
```

由于在mainDaemon（ ）中创建serveapi这个Job之后，给Job添加环境变量时，环境变量BufferRequets的值为true，故使用包

listenbuffer创建listener实例。

Listenbuffer的作用是：让Docker Server立即监听指定协议地址上的请求，但是将这些请求暂时先缓存下来，等Docker Daemon全部启动完毕之后，才让Docker Server开始接受这些请求。这样设计有一个很大的好处，那就是可以保证在Docker Daemon还没有完全启动完毕之前，接收并缓存尽可能多的用户请求。

若协议的类型为TCP，另外Job中环境变量Tls或者TlsVerify有一个为true，则说明Docker Server需要支持HTTPS服务，Docker需要为Docker Server配置安全传输层协议（TLS）的支持。实现TLS协议，首先需要建立一个tls.Config类型实例tlsConfig，然后在tlsConfig中加载证书、认证信息等，最终通过tls包中的NewListener函数，创建出适应于接收HTTPS协议请求的Listener实例l，代码如下：

```
l = tls.NewListener(l, tlsConfig)
```

至此，创建网络监听的Listener部分已经全部完成。

5.4.3　创建http.Server

Docker Server同样需要创建一个Server对象来运行HTTP/HTTPS服务端。在ListenAndServe实现中第三个重要的工作就是创建http.Server，实现源码如下：

```
httpSrv := http.Server{Addr: addr, Handler: r}
```

其中addr为需要监听的地址，r为mux.Router路由实例。

5.4.4　启动API服务

创建http.Server实例之后，Docker Server立即启动API服务，使Docker Server开始在Listener监听实例l上接受请求，并对于每一个请求都生成一个新的协程来做专属服务。对于每一个请求，协程会读取请求，查询路由表中的路由记录项，找到匹配的路由记录，最终调用路由记录中的处理方法，执行完毕后，协程对请求返回响应信息。代码如下：

```
return httpSrv.Serve(l)
```

至此，ListenAndServer的所有流程已经分析完毕，Docker Server已经开始针对不同的协议，服务API请求。

5.5 总结

Docker Server作为Docker Daemon架构中请求的入口，接管了Docker Client与Docker Daemon之间所有的通信。通信API的规范性，通信过程的安全性，服务请求的并发能力，往往都是Docker用户最为关心的内容。本章基于Docker源码，分析了Docker Server大部分的细节实现。希望Docker用户可以初探Docker Server的设计理念，并且可以更好地利用Docker Server创造更大的价值。

第6章　Docker Daemon网络

6.1　引言

Docker作为一个开源的轻量级虚拟化容器引擎技术，已然给云计算领域带来全新的发展模式。Docker借助容器技术彻底释放了轻量级虚拟化技术的威力，让容器的伸缩、应用的运行都变得前所未有的方便与高效。同时，Docker借助强大的镜像技术，让应用的分发、部署与管理变得史无前例的便捷。然而，Docker毕竟是一项较为新颖的技术，在Docker的世界中，用户并非一劳永逸，其中最为业界所诟病的便是Docker的网络问题。

毋庸置疑，对于Docker管理者和开发者而言，如何有效、高效地管理Docker容器之间的交互以及Docker容器的网络一直是一个巨大的挑战。目前，云计算领域中，绝大多数系统都采取分布式技术来设计并实现。然而，在原生态的Docker世界中，Docker的网络却不具备跨宿主机的能力，这也或多或少制约着Docker在云计算领域的高速发展。

Docker网络问题的解决势在必行，面对不同的应用场景，不少IT企业都开发了各自的新产品来帮助完善Docker的网络。这些企业中不

乏像Google一样的互联网翘楚，同时也有不少初创企业率先出击，在最前沿不懈探索。这些新产品包括：Google推出的容器管理和编排工具Kubernetes，Zett.io公司开发的通过虚拟网络连接跨宿主机容器的工具Weave，CoreOS团队针对Kubernetes设计的网络覆盖工具Flannel，Docker官方的工程师Jérôme Petazzoni自己设计的SDN网络解决方案Pipework，以及SocketPlane项目 等。

对于Docker管理者与开发者而言，虽然Docker 的跨宿主机通信能力暂时不够完善，但了解Docker自身的网络架构也是很有必要的。只有深入了解Docker自身的网络设计与实现，才能清楚其弊端，从而扩展Docker的跨宿主机能力。

Docker自身的网络主要包含两部分：Docker Daemon的网络配置、Docker容器的网络配置。本章主要从源码的角度，分析Docker Daemon在启动过程中，为Docker配置的网络环境，内容安排如下：

1) Docker Daemon网络配置。

2) 运行Docker Daemon网络初始化任务。

3) 创建Docker网桥。

6.2　Docker Daemon网络介绍

在Docker环境中，Docker容器的网络能力一直备受关心。对一个Docker容器而言，它可以独享内部的网络栈，并与其他容器分处隔离的网络环境。然而作为DockerDaemon创建的容器，容器与宿主机之外建立通信时，仍然无可避免地通过宿主机物理网卡或者虚拟机的eth0虚拟网卡。既然如此，那么如何构建Docker容器与宿主机之间的网络拓扑，将是一个学习Docker网络时必须理解的关键点。

在一台没有安装Docker的宿主机上，网络环境很有可能平淡无奇。然而，当用户开始安装并启动Docker时，一切发生了变化。而Docker管理员完全有权限在启动Docker时配置Docker Daemon的网络模式。

关于Docker的网络模式，大家最熟知的应该就是“桥接”模式，也被称为bridge模式。在桥接模式下，Docker的网络环境拓扑（包括Docker Daemon网络环境和Docker Container网络环境）如图6-1所示。

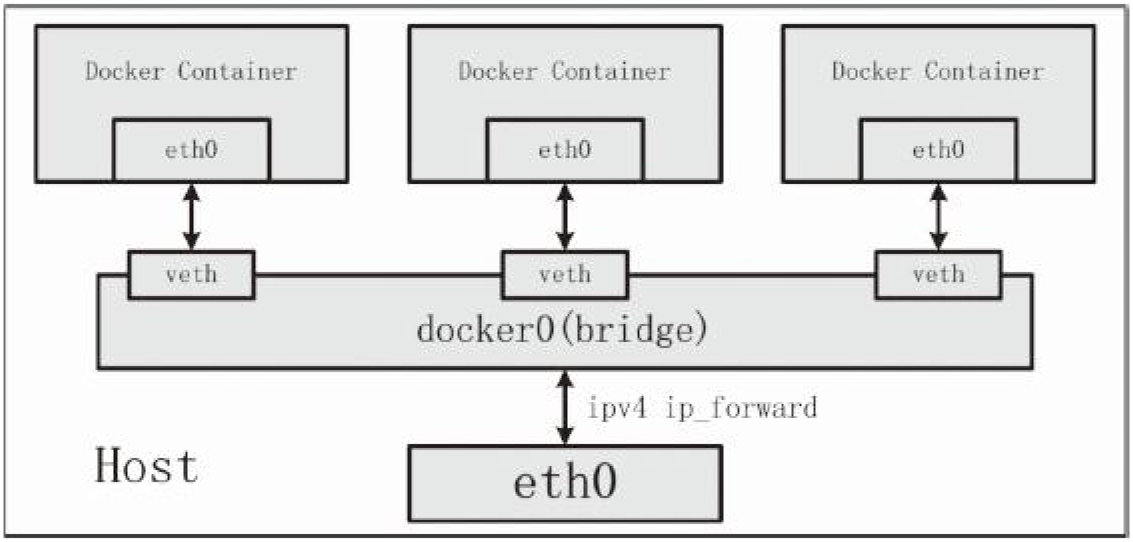


图6-1　Docker网络桥接模式示意图

然而，“桥接”模式是Docker网络模式中最为常用的模式。除此之外，Docker还为用户提供了更多的可选项，本章余下部分将进行深入分析。

6.3 Docker Daemon网络配置接口

Docker Daemon每次启动的过程中，都会初始化自身的网络环境。初始化后的网络环境最终为Docker容器提供网络通信服务。为了实现Docker Daemon网络的初始化，Docker管理员可以在启动Docker Daemon时，通过参数的形式配置Docker的网络环境。配置参数可以通过运行docker二进制可执行文件来完成，即通过运行docker-d并添加其他与网络相关的flag参数来完成。

其中，涉及的flag参数有EnableIptables、EnableIpForward、BridgeIface、BridgeIP以及InterContainerCommunication。这5个与网络相关的flag参数的定义位于./docker/docker/daemon/config.go#L49-L53，具体源码如下：

```
flag.BoolVar(&config.EnableIptables, []string{"#iptables", "-iptables"}, true,
"Enable Docker's addition of iptables rules")
flag.BoolVar(&config.EnableIpForward, []string{"#ip-forward", "-ip-forward"},
true, "Enable net.ipv4.ip_forward")
flag.StringVar(&config.BridgeIP, []string{"#bip", "-bip"}, "", "Use this CIDR
notation address for the network bridge's IP, not compatible with -b")
flag.StringVar(&config.BridgeIface, []string{"b", "-bridge"}, "", "Attach
containers to a pre-existing network bridge\nuse 'none' to disable container
networking")
flag.BoolVar(&config.InterContainerCommunication, []string{"#icc", "-icc"}, true,
"Enable inter-container communication")
```

以下介绍这5个flag参数的作用：

·EnableIptables：确保Docker Daemon启动时，能对宿主机上的iptables规则进行修改。

·EnableIpForward：确保net.ipv4.ip_forward功能开启，使得宿主机在多网络接口模式下，数据包可以在网络接口之间转发。

·BridgeIP：Docker Daemon为网络环境中的网桥配置的CIDR网络地址。

·BridgeIface：为Docker网络环境指定具体的通信网桥，若BridgeIface的值为none，则说明不需要为Docker容器创建网桥服务，关闭Docker容器的网络能力。

·InterContainerCommunication：确保Docker容器之间可以完成通信，通过防火墙完成。

除DockerDaemon会使用到的5个flag参数之外，Docker在创建网络环境时，还使用一个DefaultIP变量，如下所示：

```
opts.IPVar(&config.DefaultIp, []string{"#ip", "-ip"}, "0.0.0.0", "Default IP
address to use when binding container ports")
```

该变量的作用是：绑定Docker容器的端口与宿主机上的某一个端口时，将DefaultIp作为默认使用的宿主机IP地址。

具备以上Docker Daemon的网络背景知识之后，我们来分析如何通过docker二进制文件启动Docker Dameon并配置相应的网络环境。Docker Daemon的网络环境和两个flag参数相关性最大，分别为BridgeIP和BridgeIface。举例说明如表6-1所示。

表6-1　Docker Daemon启动命令表

启动 Docker Daemon 命令	作用分析
docker -d	启动 Docker Daemon，使用默认网桥 docker0，不指定 CIDR 网络地址
docker -d -b="xxx"	启动 Docker Daemon，使用网桥 xxx，不指定 CIDR 网络地址
docker -d --bip="172.17.42.1"	启动 Docker Daemon，使用默认网桥 docker0，使用指定 CIDR 网络地址 172.17.42.1
docker -d --bridge="xxx" --bip="10.0.42.1"	报错，出现兼容性问题，不能同时指定 BridgeIP 和 BridgeIface
docker -d --bridge="none"	启动 Docker Daemon，不创建 Docker 网络环境

深入理解BridgeIface与BridgeIP，并熟练使用相应的flag参数，就能做到配置Docker Daemon的网络环境。需要特别注意的是，Docker Daemon的网络与Docker容器的网络存在很大的区别。Docker Daemon为Docker容器创建网络的大环境，Docker容器的网络需要Docker Daemon的网络提供支持，但不唯一。举一个形象的例子，Docker Daemon可以创建docker0网桥，为之后Docker容器的桥接模式提供支持；然而在桥接模式下，Docker容器可以启用桥接，转而根据用户需求创建自身网络。其中Docker容器的网络可以是桥接模式的网络，同时也可以直接共享宿主机的网络设备，另外还有其他模式。关于Docker容器的网络，将在第7章进行详细介绍。

6.4　Docker Daemon网络初始化

正如上一节所言，Docker管理员可以通过与网络相关的flag参数BridgeIface与BridgeIP，来为Docker Daemon创建网络环境。最简单的，Docker管理员通过执行"docker-d"就已经完成Docker Daemon的运行。

Docker Daemon网络初始化流程如图6-2所示。

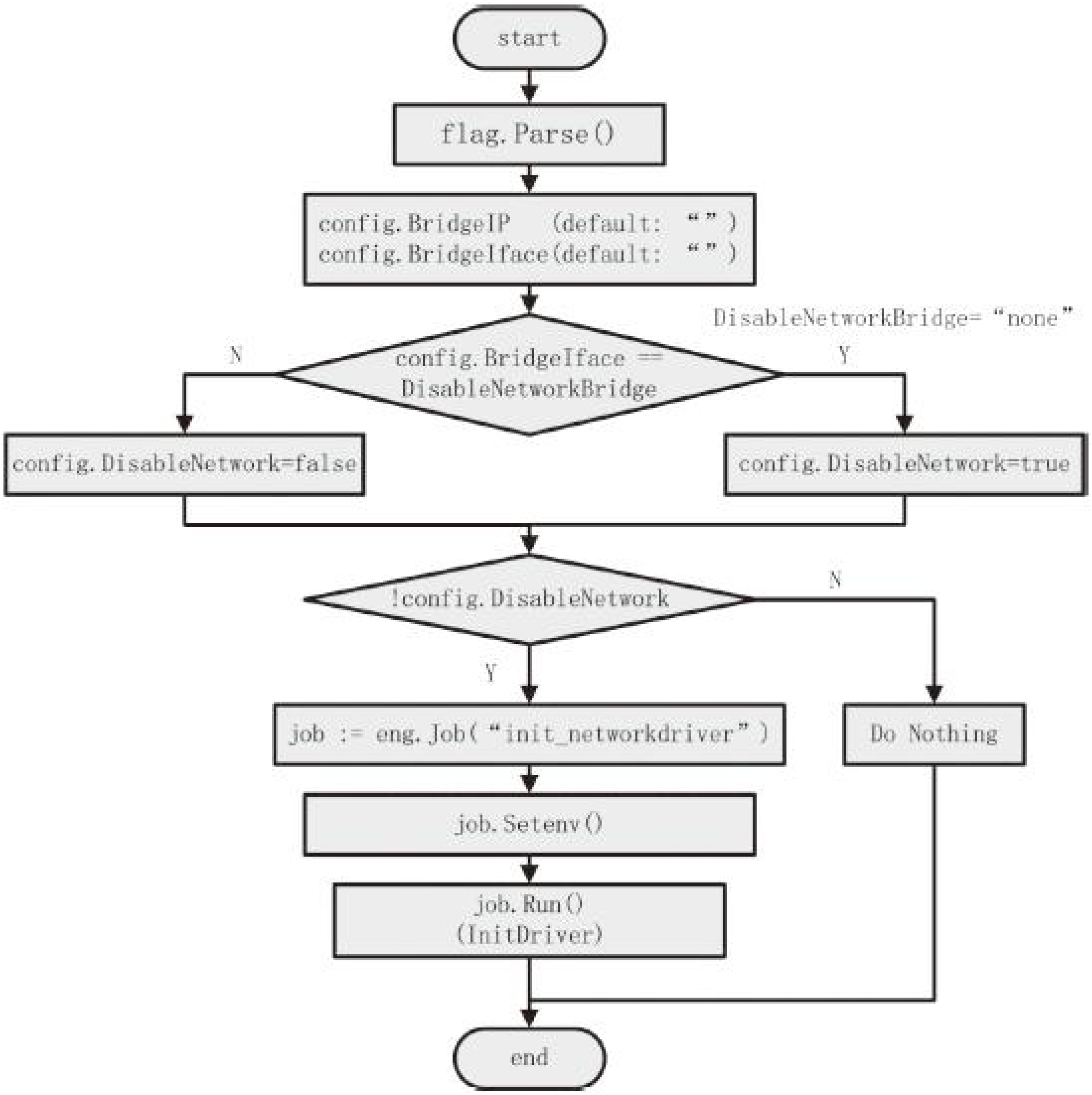


图6-2　Docker Daemon网络初始化流程图

总体而言，Docker Daemon网络的初始化流程主要是根据解析flag参数来决定到底建立哪种类型的网络环境。从流程图中可知，Docker Daemon创建网络环境时有两个分支，不难发现分支代表的分别是：为Docker创建一个网络驱动，以及对Docker的网络不做任何操作。

下面参照Docker Daemon网络初始化流程图具体分析实现步骤。

6.4.1　启动Docker Daemon传递flag参数

用户希望感受到Docker带来的便利，首先必须要有一个运行着的Docker Daemon。因此，用户第一步要完成的就是启动Docker Daemon。用户可以通过docker可执行文件来启动Docker Daemon，并在命令行中选择性地传入所需要的flag参数。

6.4.2　解析网络flag参数

Docker Daemon的启动前期，Docker使用flag包对命令行中的flag参数进行解析。其中和Docker Daemon网络配置相关的flag参数有五个，分别是：EnableIptables、EnableIpForward、BridgeIP、BridgeIface以及InterContanierCommunication，各个flag参数的作用本章已有介绍。

6.4.3　预处理flag参数

解析完与Docker Daemon网络相关的flag参数之后，Docker仍然需要对这些参数的值进行预处理。预处理与网络配置相关的flag参数信息，包括检测配置信息的兼容性，以及判断是否创建Docker网络环境。

首先，Docker需要检验flag参数中是否会出现彼此不兼容的信息，源码位于./docker/docker/daemon/daemon.go#L679-L685，如下所示：

```
// Check for mutually incompatible config options
if config.BridgeIface != "" && config.BridgeIP != "" {
          return nil, fmt.Errorf("You specified -b & --bip, mutually exclusive options. Please specify only one.")
}
if !config.EnableIptables && !config.InterContainerCommunication {
          return nil, fmt.Errorf("You specified --iptables=false with --icc=false. ICC uses iptables to function. Please set --icc or --iptables to true.")
}
```

flag参数的兼容信息主要有两种。

第一种，BridgeIP和BridgeIface配置信息的兼容性。具体表现为用户启动Docker Daemon时，若同时指定了BridgeIP和BridgIface的值，则出现兼容问题。原因在于这两者属于互斥对，换言之，若用户指定了新建网桥的设备名，那么该网桥已经存在，无须

指定网桥的IP地址BridgeIP；若用户指定了新建网桥的网络IP地址BridgeIP，那么该网桥肯定还没有新建成功，则Docker Daemon在新建网桥时使用默认网桥名docker0，无须在另行指定网桥的设备名。

第二种，EnableIptables和InterContainerCommunication配置信息的兼容性。具体表现为不能同时指定这两个flag参数为false。原因很简单，若指定InterContainerCommunication为false，则说明Docker Daemon不允许创建的Docker容器之间进行互相通信。但是为了达到以上目 的，Docker正是使用iptables防火墙的过滤规则。因此，再次设定EnableIptables为false，关闭iptables的使用，即出现了自相矛盾的结果。

检验完系统配置信息的兼容性问题，Docker Daemon接着会判断是否需要为Docker Daemon配置网络环境。判断的依据为BridgeIface的值是否与DisableNetworkBridge的值相等，DisableNetworkBridge在./docker/docker/daemon/config.go#L13中被定义为const常量，值为字符串none。因此，若BridgeIface为none，则DisableNetwork为true，最终Docker Daemon不会创建网络环境；若BridgeIface不为none，则DisableNetwork为false，最终Docker Daemon需要创建网络环境(桥接模式)。

6.4.4　确定Docker网络模式

Docker网络模式由配置信息DisableNetwork决定。由于在上一节Docker Daemon已经得出DisableNetwork的值，故本节可以确定Docker网络模式。这部分的源码实现位于./docker/docker/daemon/daemon.go#L792-L805，如下所示：

```
if !config.DisableNetwork {
     job := eng.Job("init_networkdriver")
     job.SetenvBool("EnableIptables", config.EnableIptables)
          job.SetenvBool("InterContainerCommunication",
config.InterContainerCommunication)
     job.SetenvBool("EnableIpForward", config.EnableIpForward)
     job.Setenv("BridgeIface", config.BridgeIface)
     job.Setenv("BridgeIP", config.BridgeIP)
     job.Setenv("DefaultBindingIP", config.DefaultIp.String())
     if err := job.Run(); err != nil {
            return nil, err
     }
}
```

若DisableNetwork为false，则说明需要为Docker Daemon创建网络环境，具体的模式为网桥模式。创建Docker Daemon网络环境的步骤为：

1）创建名为init_networkdriver的Job。

2）为Job配置环境变量，配置的环境变量有EnableIptables、InterContainerCommunication、EnableIpForward、

BridgeIface、BridgeIP以及DefaultBindingIP。

3）触发执行Job。

运行init_network实际完成的工作是创建Docker网桥，这部分内容将会在下一节详细分析。

若DisableNetwork为true。则说明不需要为Docker Daemon创建网络环境，网络模式属于none模式。

以上便是Docker Daemon网络初始化的所有流程。

6.5　创建Docker网桥

Docker的网络往往是Docker开发者最常提起的话题。而Docker网络中最常使用的模式为桥接模式。本节将详细分析Docker网桥的创建流程。

Docker网桥的创建通过init_network这个Job的运行来完成。init_network的实现为InitDriver函数，位于./docker/docker/daemon/networkdriver/bridge/driver.go#L79，InitDriver函数的运行流程如图6-3所示。

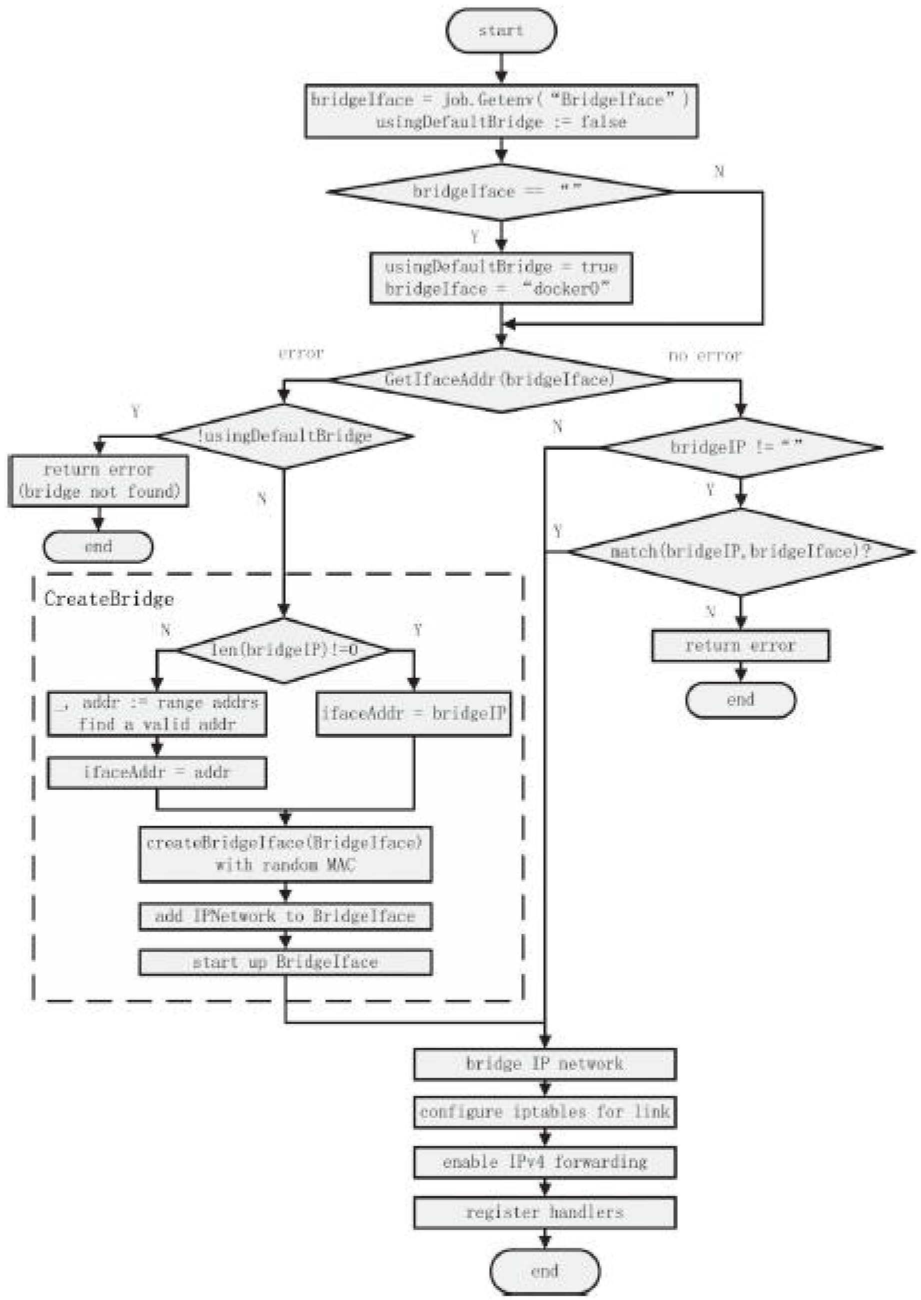


图6-3　Docker Daemon创建网桥流程图

6.5.1 提取环境变量

在InitDriver函数的实现过程中，Docker首先提取init_networkdriver这个Job的环境变量。这样的环境变量共有6个，各自的作用在上文已经详细说明。具体的实现源码如下：

```
var (

     network        *net.IPNet

     enableIPTables = job.GetenvBool("EnableIptables") icc            =
job.GetenvBool("InterContainerCommunication") ipForward      =
job.GetenvBool("EnableIpForward") bridgeIP       = job.Getenv("BridgeIP") )

if defaultIP := job.Getenv("DefaultBindingIP"); defaultIP != "" {

     defaultBindingIP = net.ParseIP(defaultIP) }

bridgeIface = job.Getenv("BridgeIface")
```

6.5.2　确定Docker网桥设备名

提取Job的环境变量之后，Docker随即确定最终使用网桥设备的名称。为此，Docker首先创建了一个名为usingDefaultBridge的bool变量，含义为是否使用默认的网桥设备，默认值为false。接着，若环境变量中bridgeIface的值为空，则说明用户启动Docker时，没有指定特定的网桥设备名，因此Docker首先将usingDefaultBridge置为true，然后使用默认的网桥设备名DefaultNetworkBridge赋值于bridgeIface，即docker0；若bridgeIface的值不为空，则判断条件不成立，继续往下执行。这部分的源码实现为：

```
usingDefaultBridge := false if bridgeIface == "" {

    usingDefaultBridge = true bridgeIface = DefaultNetworkBridge }
```

6.5.3　查找bridgeIface网桥设备

确定Docker网桥设备名bridgeIface之后，Docker首先通过bridgeIface设备名在宿主机上查找该设备是否真实存在。若存在，则返回该网桥设备的IP地址，若不存在，则返回nil。实现源码位于./docker/docker/daemon/networkdriver/bridge/driver.go#L99，如下所示：

```
addr, err := networkdriver.GetIfaceAddr(bridgeIface)
```

GetIfaceAddr的实现位于./docker/docker/daemon/networkdriver/utils.go，实现步骤为：首先通过Golang中net包的InterfaceByName方法获取名为bridgeIface的网桥设备，会得出以下结果：

·若名为bridgeIface的网桥设备不存在，直接返回错误。

·若名为bridgeIface的网桥设备存在，返回该网桥设备的IP地址。

需要强调的是：GetIfaceAddr函数返回错误，说明当前宿主机上不存在名为bridgeIface的网桥设备。而这样的结果会有两种不同的情况：第一，用户指定了bridgeIface，那么usingDefaultBridge

为false，而该bridgeIface网桥设备在宿主机上不存在；第二，用户没有指定bridgeIface，那么usingDefaultBridge为true，bridgeIface名为docker0，而docker0网桥在宿主机上也不存在。

当然，若GetIfaceAddr函数返回的是一个IP地址，则说明当前宿主机上存在名为bridgeIface的网桥设备。这样的结果同样会有两种不同的情况：第一，用户指定了bridgeIface，那么usingDefaultBridge为false，而该bridgeIface网桥设备在宿主机上已经存在；第二，用户没有指定bridgeIface，那么usingDefaultBridge为true，bridgeIface名为docker0，而docker0网桥在宿主机上也已经存在。第二种情况一般是：用户在宿主机上第一次启动Docker Daemon时，创建了默认网桥设备docker0，而后docker0网桥设备一直存在于宿主机上，故之后在不指定网桥设备的情况下，重启Docker Daemon，会出现docker0已经存在的情况。

以下两节将分别从bridgeIface已创建与bridgeIface未创建两种不同的情况进行深入分析。

6.5.4 bridgeIface已创建

Docker Daemon所在宿主机上bridgeIface的网桥设备存在时，Docker Daemon仍然需要验证用户在配置信息中是否为网桥设备指定了IP地址。

用户启动Docker Daemon时，假如没有指定bridgeIP参数信息，则Docker Daemon使用名为bridgeIface的原有的IP地址。

当用户指定了bridgeIP参数信息时，则需要验证：指定的bridgeIP参数信息与bridgeIface网桥设备原有的IP地址信息是否匹配。若两者匹配，则验证通过，继续往下执行；若两者不匹配，则验证不通过，抛出错误，显示“bridgeIP与已有网桥配置信息不匹配”。该部分内容位于./docker/docker/daemon/networkdriver/bridge/driver.go#L119-L129，源码如下：

```
network = addr.(*net.IPNet)
// validate that the bridge ip matches the ip specified by BridgeIP
if bridgeIP != "" {
     bip, _, err := net.ParseCIDR(bridgeIP)
     if err != nil {
          return job.Error(err)
     }
     if !network.IP.Equal(bip) {
               return job.Errorf("bridge ip (%s) does not match existing bridge
configuration %s", network.IP, bip)
     }
}
```

6.5.5 bridgeIface未创建

Docker Daemon所在宿主机上bridgeIface的网桥设备未创建时，上文已经介绍将存在两种情况：

·用户指定的bridgeIface未创建。

·用户未指定bridgeIface，而docker0暂未创建。

当用户指定的bridgeIface不存在于宿主机时，即没有使用Docker的默认网桥设备名docker0，Docker打印日志信息“指定网桥设备未找到”，并返回网桥未找到的错误信息。源码实现如下：

```
if !usingDefaultBridge {
     job.Logf("bridge not found: %s", bridgeIface)
     return job.Error(err)
}
```

当使用默认网桥设备名，而docker0网桥设备还未创建时，Docker Daemon则立即实现创建网桥的操作，并返回该docker0网桥设备的IP地址。代码如下：

```
// If the iface is not found, try to create it
job.Logf("creating new bridge for %s", bridgeIface)
if err := createBridge(bridgeIP); err != nil {
     return job.Error(err)
}
job.Logf("getting iface addr")
addr, err = networkdriver.GetIfaceAddr(bridgeIface)
if err != nil {
     return job.Error(err)
```

```
}
network = addr.(*net.IPNet)
```

创建Docker Daemon网桥设备docker0的实现，全部由createBridge（bridgeIP）来实现，createBridge的实现位于./docker/docker/daemon/networkdriver/bridge/driver.go#L245。createBridge函数实现步骤如下：

1）确定网桥设备docker0的IP地址。

2）通过createBridgeIface函数创建docker0网桥设备，并为网桥设备分配随机的MAC地址。

3）将第一步中已经确定的IP地址，添加给新创建的docker0网桥设备。

4）启动docker0网桥设备。

下面详细分析4个步骤的具体实现。

第一步，Docker Daemon确定docker0的IP地址，实现方式为判断用户是否为网桥设备指定bridgeIP。若用户未指定bridgeIP，则从Docker预先准备的IP网段列表addrs中查找合适的网段。具体的源码实现位于./docker/docker/daemon/networkdriver/bridge/driver.go#L257-L278，如下所示：

```
if len(bridgeIP) != 0 {
     _, _, err := net.ParseCIDR(bridgeIP)
     if err != nil {
          return err
     }
     ifaceAddr = bridgeIP
} else {
     for _, addr := range addrs {
          _, dockerNetwork, err := net.ParseCIDR(addr)
          if err != nil {
               return err
          }
               if err := networkdriver.CheckNameserverOverlaps(nameservers,
dockerNetwork); err == nil {
               if err := networkdriver.CheckRouteOverlaps(dockerNetwork); err ==
nil {
                    ifaceAddr = addr
                    break
               } else {
                    log.Debugf("%s %s", addr, err)
               }
          }
     }
}
```

其中，Docker Daemon为网桥设备准备的候选网段地址addrs为：

```
addrs = []string{
          "172.17.42.1/16", // Don't use 172.16.0.0/16, it conflicts with EC2 DNS
172.16.0.23
     "10.0.42.1/16", // Don't even try using the entire /8, that's too intrusive
     "10.1.42.1/16",
     "10.42.42.1/16",
     "172.16.42.1/24",
     "172.16.43.1/24",
     "172.16.44.1/24",
     "10.0.42.1/24",
     "10.0.43.1/24",
     "192.168.42.1/24",
     "192.168.43.1/24",
     "192.168.44.1/24",
}
```

通过执行以上流程，Docker Daemon可以确定一个可用的IP网段地址，为ifaceAddr；若仍然未匹配合适的网段地址，

DockerDaemon返回错误日志，表明没有合适的IP地址赋予docker0网桥设备。

第二步，DockerDaemon通过createBridgeIface函数创建docker0网桥设备。createBridgeIface函数的源码实现如下：

```
func createBridgeIface(name string) error {
     kv, err := kernel.GetKernelVersion()
     // only set the bridge's mac address if the kernel version is > 3.3
     // before that it was not supported
     setBridgeMacAddr := err == nil && (kv.Kernel >= 3 && kv.Major >= 3)
     log.Debugf("setting bridge mac address = %v", setBridgeMacAddr)
     return netlink.CreateBridge(name, setBridgeMacAddr)
}
```

以上代码通过宿主机Linux内核信息，确定宿主机内核版本是否支持设定网桥设备的MAC地址。若Linux内核版本大于3.3，则支持配置MAC地址，否则不支持。而Docker在不低于3.8的内核版本上才能稳定运行，故可以认为内核支持配置MAC地址。最后通过netlink的CreateBridge函数实现创建docker0网桥。

Netlink是Linux中一种较为特殊的socket通信方式，提供了用户应用间和内核进行双向数据传输的途径。在这种模式下，用户态可以使用标准的socket API来使用netlink强大的功能，而内核态需要使用专门的内核API才能使用netlink。

libcontainer的netlink包中的CreateBridge实现了创建实际的网桥设备，具体使用系统调用的代码如下：

```
syscall.Syscall(syscall.SYS_IOCTL, uintptr(s), SIOC_BRADDBR,
uintptr(unsafe.Pointer(nameBytePtr)))
```

创建完网桥设备之后，为docker0网桥设备配置MAC地址，实现函数为setBridge-MacAddress。

第三步为创建的docker0网桥设备绑定IP地址。上一步仅完成了创建名为docker0的网桥设备，之后仍需要为docker0网桥设备绑定IP地址。具体源码实现为：

```
if netlink.NetworkLinkAddIp(iface, ipAddr, ipNet); err != nil {
     return fmt.Errorf("Unable to add private network: %s", err)
}
```

NetworkLinkAddIP的实现同样位于libcontainer中的netlink包，主要的功能为：通过netlink机制为一个网络接口设备绑定一个IP地址。

第四步是启动docker0网桥设备。具体实现代码如下：

```
if err := netlink.NetworkLinkUp(iface); err != nil {
     return fmt.Errorf("Unable to start network bridge: %s", err)
}
```

NetworkLinkUp的实现同样位于libcontainer中的netlink包，功能为启动docker0网桥设备。

至此，docker0网桥历经确定IP、创建、绑定IP、启动四个环节，createBridge关于docker0网桥设备的工作全部完成。

6.5.6 获取网桥设备的网络地址

创建完网桥设备之后，网桥设备必然会存在一个网络地址。网桥网络地址的作用为：Docker Daemon在创建Docker容器时，使用该网络地址为Docker容器分配IP地址。Docker使用源码network= addr.(*net.IPNet) 获取网桥设备的网络地址。

6.5.7　配置Docker Daemon的iptables

创建完网桥之后，Docker Daemon为容器以及宿主机配置iptables，包括为容器之间所需要的link操作提供支持，为宿主机上所有的对外对内流量制定传输规则等。这部分详情可参见第4章，源码实现位于./docker/daemon/networkdriver/bridge/driver/driver.go#L133，如下所示：

```
// Configure iptables for link support if enableIPTables {

     if err := setupIPTables(addr, icc); err != nil {

          return job.Error(err) }

}

// We can always try removing the iptables if err :=
iptables.RemoveExistingChain("DOCKER"); err != nil {

     return job.Error(err) }

if enableIPTables {

     chain, err := iptables.NewChain("DOCKER", bridgeIface) if err != nil {

          return job.Error(err) }
```

```
portmapper.SetIptablesChain(chain) }
```

6.5.8　配置网络设备间数据报转发功能

默认情况下，Linux操作系统上的数据包转发功能是禁止的。数据包转发就是当宿主机存在多个网络设备时，如果其中一个接收到数据包，并将其转发给另外的网络设备。Docker Daemon通过修改/proc/sys/net/ipv4/ip_forward的值，将其置为1，则可以保证系统内数据包可以实现转发功能，代码如下：

```
if ipForward {

     // Enable IPv4 forwarding

          if err := ioutil.WriteFile("/proc/sys/net/ipv4/ip_forward", []byte{'1', '\n'}, 0644); err != nil {

          job.Logf("WARNING: unable to enable IPv4 forwarding: %s\n", err)

     }

}
```

6.5.9　注册网络Handler

创建Docker Daemon网络环境的最后一个步骤是：注册4个与网络相关的Handler。这4个处理方法分别是allocate_interface、release_interface、allocate_port和link，作用分别是为Docker容器分配IP网络地址，释放Docker容器网络设备，为Docker容器分配端口资源，以及在Docker容器之间执行link操作。

至此，Docker Daemon的网络环境初始化工作全部完成。

6.6　总结

在工业界，Docker的网络问题备受关注。Docker的容器技术以及镜像技术，已经给Docker实践者带来了诸多效益。然而Docker网络的发展依然具有很大的潜力，依然值得大家不断探索并推动其发展。

Docker的网络环境可以分为Docker Daemon网络和Docker Container网络。本章从Docker Daemon的网络入手，分析了大家熟知的Docker桥接模式。

第7章　Docker容器网络

7.1　引言

如今，Docker技术风靡全球，大家在尝试以及玩转Docker的同时，肯定离不开一个概念，那就是“容器”或者“Container”。那么我们首先从原理的角度一窥“容器”或者“Container”的究竟。

第一次接触Docker时，大家肯定会深深感受到：在Docker容器内部运行应用程序竟是如此方便。第一，应用程序在Docker容器内部的部署与运行非常便捷，只要有Dockerfile，应用一键式的部署绝对不是天方夜谭；第二，Docker容器内运行的应用程序还可以受到资源的控制与隔离，大大满足云计算时代应用的要求。

毋庸置疑，Docker的一些特性，的确摒弃了传统情况下开发模式的不少弊端。然而，这强大的功能背后到底是何等技术在“作祟”，又是谁可以支撑Docker的运行并提供丰富的特性？答案很简单，那就Linux内核。

那我们就从Linux内核的角度来看看Docker到底为何物，先从Docker容器入手。对于Docker容器，体验过的开发者都有两点非常

重要的感受：内部可以跑应用(进程)，以及提供隔离的环境。当然，后者肯定也是工业界称之为“容器”的原因之一。

首先，我们先来看Docker容器与进程的关系，或者容器与进程的关系。为了理清这两者之间的关系，我不妨提出这样一个问题“容器是否可以脱离进程而存在”。换句话说就是，能否创建一个容器，而这个容器内部没有任何进程。答案是否定的。既然答案是否定的，说明不可能先有容器，然后再有进程，那么问题又来了，“容器和进程是一起诞生的，还是先有进程再有容器呢？”可以说答案是后者。以下将慢慢阐述其中的原因。

阐述问题“容器是否可以脱离进程而存在”的原因前，相信大家对于下面这段话不会有异议：通过Docker创建出的一个Docker Container是一个容器，而这个容器提供了进程组隔离的运行环境。那么问题在于，容器到底是通过什么途径来实现进程组运行环境的“隔离”呢？此时，就轮到Linux内核隆重登场了。

说到运行环境的“隔离”，相信大家肯定对Linux内核中的namespaces和cgroups略有耳闻，namespaces主要负责命名空间的隔离，cgroups主要负责资源使用的限制。其实，正是这两个神奇的内核特性联合使用，才保证了Docker容器之间的“隔离”。那么，namespaces和cgroups又和进程有什么关系呢？问题的答案可以用以下的次序来表示：

1） 父进程通过fork创建子进程时，使用namespaces技术，实现子进程与父进程以及其他进程之间命名空间的隔离。

2） 子进程创建完毕之后，使用cgroups技术来处理进程，实现进程的资源限制。

3） namespaces和cgroups这两种技术都用上之后，进程所处的“隔离”环境才真正建立，此时“容器”真正诞生!

从Linux内核的角度分析容器的诞生，精简的流程即如上三步，而这三个步骤也恰好巧妙地阐述了namespaces和cgroups这两种技术和进程的关系，以及进程与容器的关系。进程与容器的关系，自然是：容器不能脱离进程而存在，先有进程，后有容器。然而，大家往往会说到“使用Docker创建Docker容器，然后在容器内部运行进程”。对此，从通俗易懂的角度来讲，这完全可以理解，因为“容器”一词的存在，本身就较为抽象。如果需要更为准确，可以表述为“使用Docker创建一个进程，为这个进程创建隔离的环境，这样的环境可以称为Docker容器，然后再在容器内部运行用户应用进程。”在这里，笔者的本意不是希望纠正外界对于Docker容器的认识，而是希望能和读者一起来看看Docker容器技术实现的原理到底几何。

更清楚地认识Docker容器之后，很快大家的眼球肯定会定位到namespaces和cgroups这两种技术上。Linux内核的这两种技术，竟然起到如此重大的作用。那么下面我们就从Docker容器实现流程的角度简要介绍这两者。

首先讲述一下namespace在容器创建时的用法。用户启动容器，Docker Daemon会fork出容器中的第一个进程A（暂且称为进程A，也就是Docker Daemon的子进程），执行fork时通过5个参数标志CLONE_NEWNS、CLONE_NEWUTS、CLONE_NEWIPC、CLONE_PID和CLONE_NEWNET（Docker 1.2.0还没有完全支持user namespace）。Clone系统调用一旦传入了这些参数标志，子进程将不再与父进程共享相同的命名空间（namespaces），而是由Linux为子进程创建新的命名空间（namespaces），从而保证子进程与父进程使用隔离的环境。另外，如果子进程A再次fork出子进程B，而fork时没有传入相应的namespaces参数标志时，子进程B将会与A共享相同的命名空间（namespaces）。如果Docker Daemon再次创建一个Docker容器，内部进程有D、E和F，那么这三个进程也会处于另外全新的namespaces中。两个容器的namespaces均与Docker Daemon所在的namespaces不同。Docker中Docker Daemon与Docker容器之间的namespaces关系，如图7-1所示。

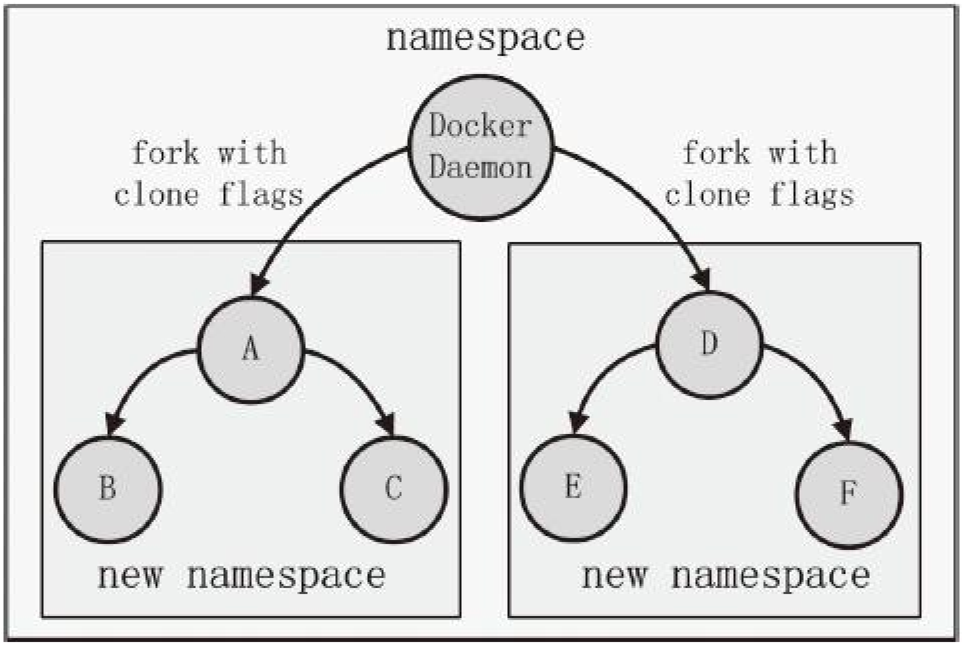


图7-1　Docker的namespaces关系图

再说起cgroups，大家都知道可以使用cgroups为进程组做资源的限制。与namespaces不同的是，cgroup的使用并不是在创建容器内进程时完成，而是在创建容器内进程之后完成，最终使得容器进程处于资源控制的状态。换言之，cgroups的运用必须要等到容器内第一个进程被真正创建出来之后才能实现。当容器内进程创建完毕，Docker Daemon可以获知容器内主进程的PID信息，随后将该PID放置在cgroups文件系统的指定位置，做相应的资源限制。如此一来，当容器主进程再fork新的子进程时，新的子进程同样受到与主进程相同的资源限制，效果就是整个进程组受到资源限制。

可以说Linux内核的namespaces和cgroups技术，实现了资源的隔离与限制。那么对于资源的隔离，是否还需要为容器准备必需的

资源，比如说容器需要使用的网络资源，容器需要使用的文件系统挂载点等。这回答案是肯定的。网络资源就是一个很好的例子。当Docker Daemon为进程创建完隔离的运行环境之后，我们可以发现进程并没有独立的网络栈可以使用，如独立的网络接口等。此时，Docker Daemon会将Docker容器内部所需要的资源一一配备齐全。网络方面，即为Docker容器通过用户指定的网络模式，配置相应的网络资源。

本章从源码的角度，分析Docker容器从无到有的过程中，Docker容器网络创建的来龙去脉。简化而言，Docker容器网络的创建流程如图7-2所示。

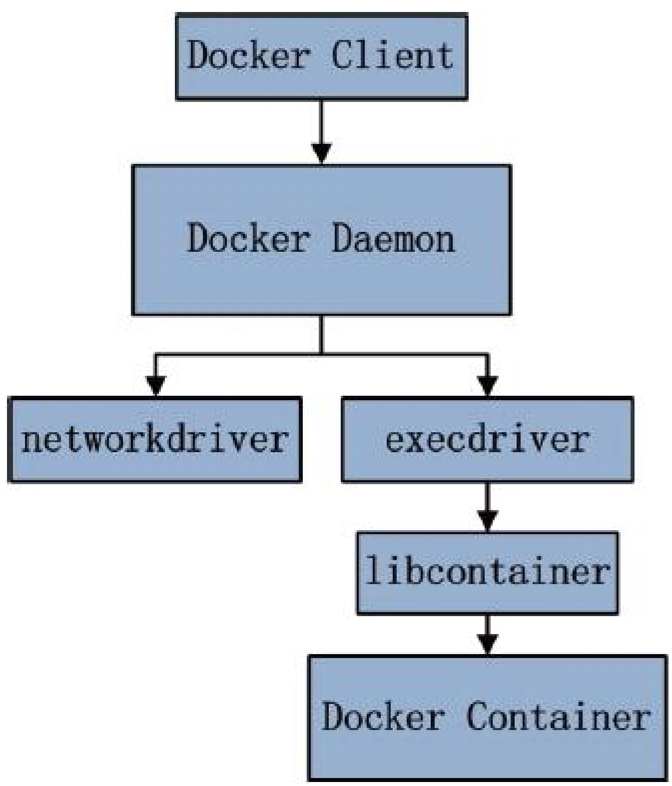


图7-2　Docker容器网络创建流程图

本章分析的主要内容有以下5部分：

·Docker容器的网络模式

·Docker Client配置容器网络

·Docker Daemon创建容器网络流程

·execdriver网络执行流程

·libcontainer实现内核态网络配置

在Docker容器的网络创建过程中，networkdriver模块使用并非是重点，故分析内容中不涉及networkdriver。不少读者肯定会有一些疑惑，为什么Docker容器的网络创建很少用到networkdriver。这里强调一下networkdriver在Docker中的作用：第一，为Docker Daemon创建网络环境的时候，初始化Docker Daemon的网络环境（详见第6章），比如创建docker0网桥等；第二，为Docker容器分配IP地址，为Docker容器做端口映射等。而与Docker容器网络创建有关的内容并不多，如只在桥接模式下，为Docker容器的网络接口设备分配一个IP地址。

7.2 Docker容器网络模式

如前所述，Docker可以为容器创建隔离的网络环境，在隔离的网络环境下，Docker容器使用独立的网络栈。看到这，很多读者也许会非常赞成以上的言论。然而，Docker容器网络的高级特性又会让爱好者们感受到其中暗藏的更多玄机。

直奔主题，其实Docker除可以为Docker容器创建隔离的网络环境之外，同样有能力为Docker容器创建共享的网络环境。换言之，当开发者需要Docker容器与宿主机或者其他容器网络隔离，Docker可以满足这样的需求；当开发者需要Docker容器的网络处于共享的网络环境时，Docker同样可以满足这样的需求，并且网络共享，可以是Docker容器与宿主机之间网络共享，也可以是Docker容器与其他容器之间网络共享。神奇的是，Docker还可以实现不为Docker容器创建网络环境。

通过以上描述，可以总结出Docker容器共有以下4种网络模式：bridge桥接模式、host模式、other container模式和none模式。下面分别介绍Docker的4种网络模式。

7.2.1　bridge桥接模式

Docker容器的bridge桥接模式是目前Docker开发者中使用最为广泛的网络模式。bridge桥接模式可以使Docker容器独立使用网络栈，或者说只有在容器内部的进程，才能使用该网络栈。bridge桥接模式的实现步骤如下。

1）利用veth pair技术，在宿主机上创建两个虚拟网络接口，假设为veth0和veth1。而veth pair技术的特性可以保证无论哪一个veth接收到网络报文，都会将报文传输给另一方。

2）Docker Daemon将veth0附加到Docker Daemon创建的docker0网桥上。保证宿主机的网络报文有能力发往veth0。

3）Docker Daemon将veth1添加到Docker容器所属的网络命名空间（namespaces）下，veth1在Docker容器看来就是eth0。一方面，保证宿主机的网络报文若发往veth0，可以立即被veth1收到，实现宿主机到Docker容器之间网络的联通性；另一方面，保证Docker容器单独使用veth1，实现容器之间以及容器与宿主机之间网络环境的隔离性。

Docker容器的bridge桥接模式如图7-3所示。

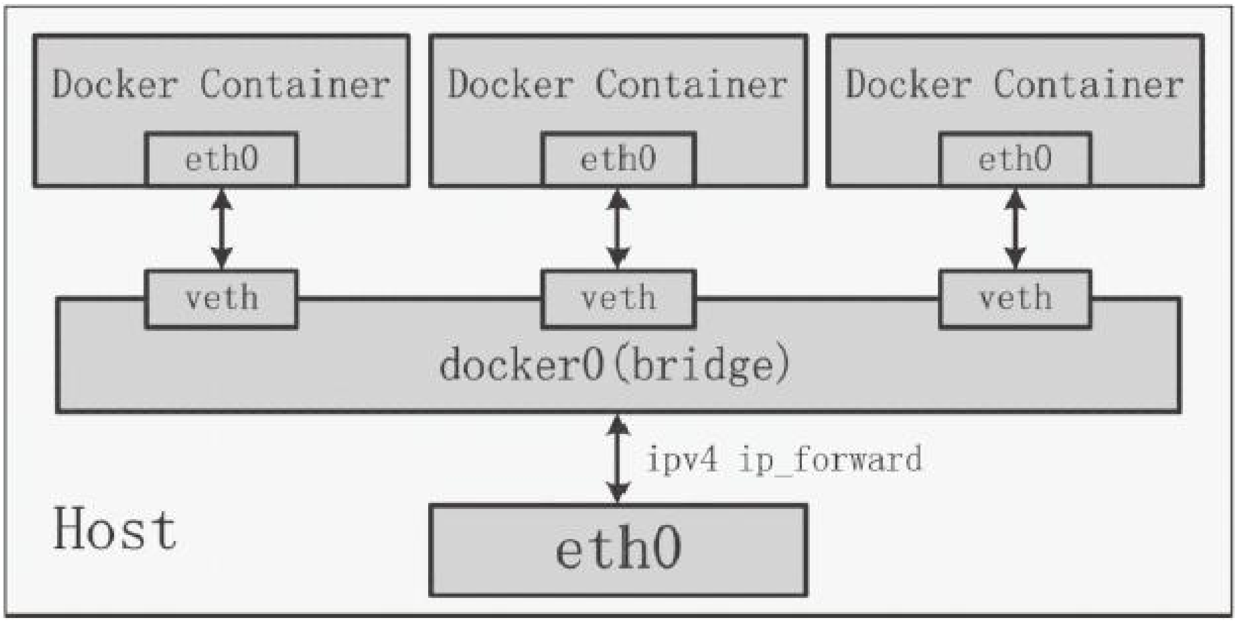


图7-3　Docker容器bridge桥接模式示意图

bridge桥接模式，从原理上实现了Docker容器到宿主机乃至其他机器的网络联通性。然而，由于宿主机的IP地址与veth pair的IP地址不属于同一个网段，故仅仅依靠veth pair和网络命名空间的技术，还不足以使宿主机以外的网络主动发现Docker容器的存在。为使Docker容器有能力让宿主机以外的世界感受到容器内部暴露的服务，Docker采用NAT（ Network Address Translation，网络地址转换）的方式让宿主机以外的世界可以将网络报文发送至容器内部。简要来讲，当Docker容器需要暴露服务时，内部服务必须监听容器IP和容器内部端口号port_0，以便外界主动发起访问请求。由于宿主机以外的世界，只知道宿主机eth0的网络地址，并不知道Docker容器的IP地址，哪怕就算知道Docker容器的IP地址，从二层网络的角度来讲，外界也无法直接通过Docker容器的IP地址访问容器内部的服务。因此，

Docker使用NAT方法，将容器内部的服务与宿主机的某一个端口port_1进行“绑定”。

如此一来，外界访问Docker容器内部服务的流程为：

1）外界访问宿主机的IP以及宿主机的端口port_1。

2）当宿主机接收到这类请求之后，由于存在DNAT规则，会将该请求的目的IP（宿主机eth0的IP）和目的端口port_1进行替换，替换为容器IP和容器端口port_0。

3）由于能够识别容器IP，故宿主机可以将请求发送给veth pair。

4）veth pair将请求发送至容器，容器交于内部服务进行处理。

使用DNAT方法，可以使Docker宿主机以外的世界主动访问Docker容器。那么Docker容器如何访问宿主机以外的世界呢？以下简要分析Docker容器内部访问宿主机以外世界的流程：

1）Docker容器内部进程获悉宿主机外部服务的IP地址和端口port_2，于是Docker容器发起请求。容器独立的网络环境保证了请求中报文的源IP地址为容器IP（即容器内部eth0，veth pair一方的IP地址），另外Linux内核会自动为进程分配一个可用端口（假设为port_3）。

2）请求通过容器内部eth0发送至veth pair的另一端，也就是到达网桥docker0处。

3）docker0网桥开启了数据报转发功能（/proc/sys/net/ipv4/ip_forward），故docker0将请求发送至宿主机的eth0处。

4）宿主机处理请求时，使用SNAT对请求进行源地址IP替换，即将请求中源地址IP（容器eth0的IP地址）替换为宿主机eth0的IP地址。

5）宿主机将经过SNAT处理后的报文通过请求的目的IP地址（宿主机以外世界的IP地址）发送至外界。

在这里，很多人肯定要问：对于Docker容器内部主动发起对外的网络请求，请求到达宿主机进行SNAT处理发给外界之后，当外界响应请求时，响应报文中的目的IP地址肯定是Docker Daemon所在宿主机的IP地址，那响应报文回到宿主机的时候，宿主机又是如何转给Docker容器的呢？关于这样的响应，由于没有做相应的DNAT转换，原则上不会被发送至容器内部。为什么说对于这样的响应，不会做DNAT转换呢。原因很简单，DNAT转换是针对特定容器内部服务监听的特定端口做的，该端口是供服务监听使用，而容器内部发起的请求

报文中，源端口号肯定不会占用服务监听的端口，故容器内部发起请求的响应不会在宿主机上经过DNAT处理。

其实，这一环节的关键在于iptables，具体的iptables规则如下：

```
iptables -I FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j
ACCEPT
```

此iptables规则的意思是：在宿主机上发往docker0网桥的网络数据报文，如果该数据报文所处的连接已经建立，则无条件接受，并由Linux内核将其转至原来的连接上，即回到Docker容器内部。

以上便是Docker容器中bridge桥接模式的简要介绍。可以说，bridger桥接模式实现了两个方面：第一，让容器内部使用独立的网络栈；第二，让容器和宿主机以外的世界通过NAT建立通信。从功能的角度来讲，恰似具备了传统情况下隔离环境中网络的功能。然而，从使用的角度来讲，这种方式还没有弥合传统网络环境与容器网络环境的缝隙，换言之，NAT方式必须使得容器服务在宿主机上有一层抽象，这层抽象需要Docker使用者人为的干涉，另外使用NAT方式，仅仅是三层网络上的实现手段，影响网络传输效率的同时，也会为网络的隔离带来不便。

7.2.2　host模式

Docker容器中的host模式与bridge桥接模式是完全不同的模式。最大的区别是：host模式并没有为容器创建一个隔离的网络环境。之所以称之为host模式，是因为该模式下的Docker容器会和host宿主机使用同一个网络命名空间，故Docker容器可以和宿主机一样，使用宿主机的eth0和外界进行通信。如此一来，Docker容器的IP地址自然也是宿主机eth0的IP地址。

Docker容器的host网络模式如图7-4所示。

图7-4中最左侧的Docker容器，即采用了host网络模式，而其他两个Docker容器依然沿用brdige桥接模式，两种模式存在于同一个宿主机上并不矛盾。

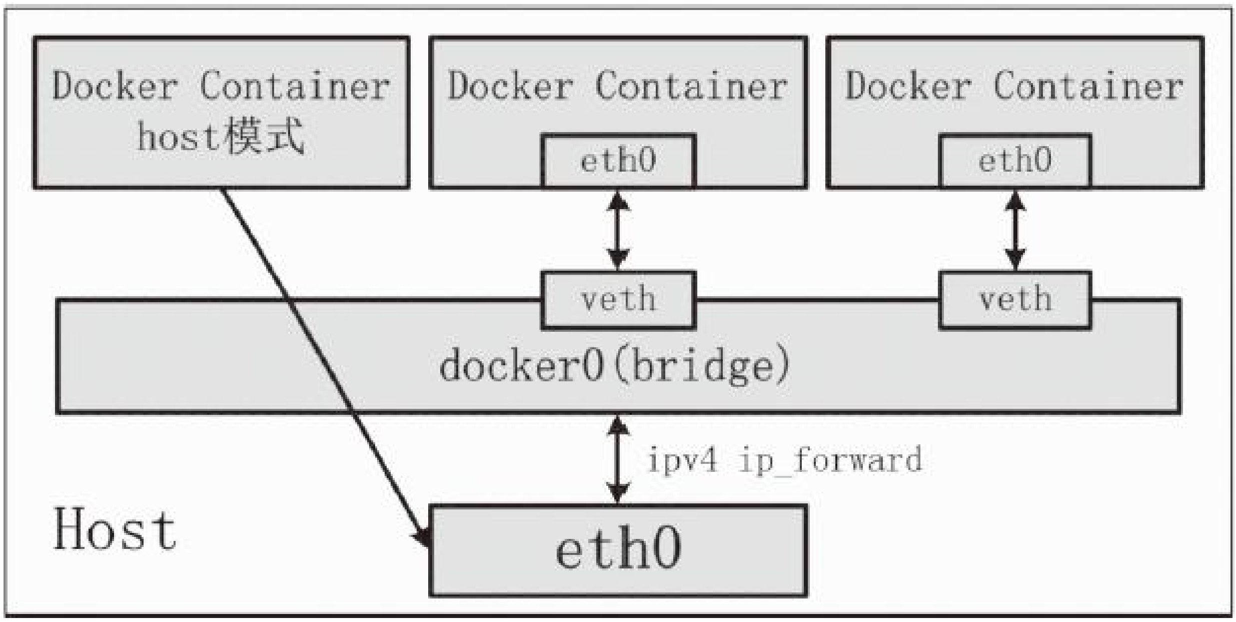


图7-4　Docker容器host网络模式示意图

Docker容器的host网络模式在实现过程中，由于不需要额外的网桥以及虚拟网卡，故不会涉及docker0以及veth pair。上文namespace的介绍中曾经提到：父进程在创建子进程的时候，如果不使用CLONE_NEWNET这个参数标志，那么创建出的子进程会与其父进程共享同一个网络命名空间。Docker就是采用了这个简单的原理，在创建进程启动容器的过程中，没有传入CLONE_NEWNET参数标志，实现Docker容器与宿主机共享同一个网络环境，网络模式即为host网络模式。

Docker容器的网络模式中，host模式是bridge桥接模式很好的补充。采用host模式的Docker容器，可以直接使用宿主机的IP地址与外界进行通信，若宿主机的eth0是一个公有IP，那么容器的IP也拥有这

个公有IP。同时容器内服务的端口也可以使用宿主机的端口，无需额外进行NAT转换。有这样的方便，自然会折损部分其他的特性，最明显的即是Docker容器网络环境的隔离性。另外，使用host模式的Docker容器虽然可以让容器内部的服务无差别、无改造的使用，但是由于网络的非隔离性，一旦宿主机上多个Docker容器都采用了host模式，并且内部的服务都需要占用相同的宿主机端口资源，此时会造成宿主机上端口资源的竞争，势必影响容器的服务质量。

7.2.3　other container模式

Docker容器的other container网络模式是Docker中一种较为特别的网络模式。之所以称为“other container模式”，是因为这个模式下的Docker容器，会使用其他容器的网络环境。之所以称为“特别”，是因为这个模式下容器的网络隔离性会处于bridge桥接模式与host模式之间。Docker容器共享其他容器的网络环境，则至少这两个容器之间不存在网络隔离，而这两个容器又与宿主机以及除此之外其他的容器存在网络隔离。

Docker容器的other container网络模式如图7-5所示。

图7-5中右侧的Docker容器即采用了other container网络模式，它具体能感受到网络环境即为左侧Docker容器的bridge桥接模式。

在Docker容器的other container网络模式实现过程中，不涉及网桥，同样也不需要创建虚拟网卡veth pair。完成other container网络的创建只需要两个步骤：

1）查找other container（即需要被共享网络环境的容器）的网络命名空间。

2）新创建的Docker容器的网络命名空间使用其他容器的网络命名空间。

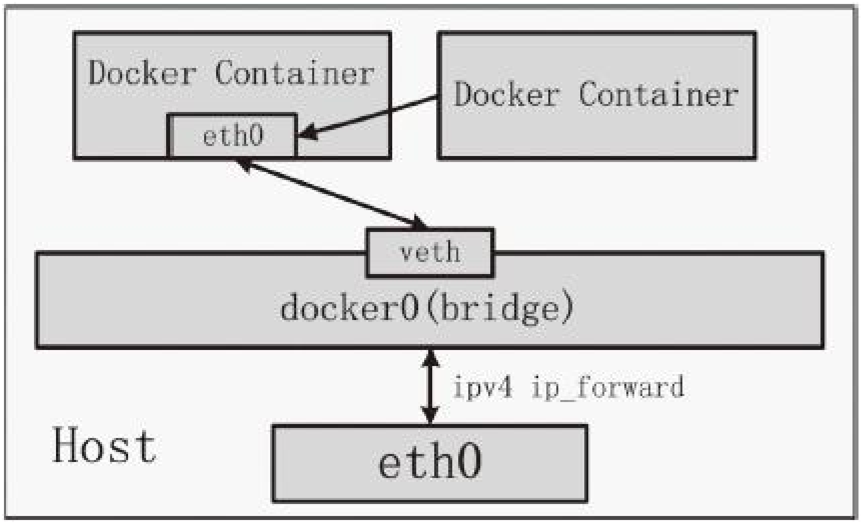


图7-5　Docker容器other container网络模式示意图

Docker容器的other container网络模式，可以用来更好地服务于容器间的通信。需要互相通信的Docker容器在这种模式下，虽然网络隔离性没有bridge桥接模式优秀，但是相比host模式自然要好不少，同时网络隔离性稍有弱化的背后，带来的却是容器间高效的传输效率，以及容器间互访时简约的网络配置，即只需要使用localhost来访问其他容器。同时，多个容器共享同一个网络命令空间的形式，使得这部分容器的粘性大大提高。在容器的大规模调度与管理中，容器组必然会是一个需要攻克的技术点，而other container网络模式似乎为容器组的实现提供了方向。值得思考的是，Google推出的

Kubernetes平台中的Pod的原理，即使用了容器间共享网络命名空间的思路，实现了组的概念。

7.2.4　none模式

Docker容器的第4种网络模式是none模式。顾名思义，网络环境为none，即不为Docker容器创建任何的网络环境。一旦Docker容器采用了none网络模式，那么容器内部就只能使用loopback网络接口，不会再有其他的网络资源。

可以说none模式为Docker容器做了极少的网络设定，但是俗话说得好“少即是多”，在没有网络配置的情况下，作为Docker开发者，才能在这种模式下做无限多可能的网络定制开发。这也恰恰体现了Docker开放的设计理念。

至此，Docker的4种网络模式的介绍就告一段落，下文带来Docker网络模式的创建流程分析。

7.3　Docker Client配置容器网络模式

Docker容器网络模式的多样性，极大地满足了Docker用户部署应用的网络需求。Docker用户，或者基于Docker的二次开发者，均可以在这基础上，选择使用与自身应用最为贴切的网络模式。

从图7-2中可以看到，Docker容器网络创建流程中第一个涉及的Docker模块就是Docker Client。当然，这也容易理解，毕竟Docker容器网络环境的创建需要由用户发起。用户根据自身对容器的需求，选择网络模式，并将其通过Docker Client传递给Docker Daemon。本小节将从Docker Client源码的角度出发，分析如何通过参数的形式配置Docker容器的网络模式，以及Docker Client内部如何处理这些网络模式参数。

7.3.1　使用Docker Client

Docker架构中，用户完全可以使用Docker Client来配置Docker Container的网络模式。实际情况下，启动一个容器，Docker才会为Docker容器创建网络环境，故配置网络模式在用户执行“启动容器”命令之后完成。启动容器命令为docker run，通过Docker Client发起，使用方式如下所示(其中NETWORKMODE为四种网络模式之一，IMAGE_NAME为镜像名称) ：

```
docker run -it--net NETWORKMODE IMAGE_NAME /bin/bash
```

执行以上命令时，Docker首先创建一个Docker Client，再由Docker Client解析整条命令的请求内容，最终解析为run命令，并发送至Docker Daemon。Docker Client的相关实现详见本书第2章。

7.3.2　runconfig包解析

解析出run命令之后，Docker Client调用相应的处理函数CmdRun处理关于run请求的具体内容。CmdRun函数的实现位于./docker/docker/api/client/commands.go#L1990。CmdRun执行的第一步就是：通过runconfig包中的ParseSubcommand函数执行解析出相应的config、hostConfig，以及cmd对象，源码实现如下：

```
config, hostConfig, cmd, err := runconfig.ParseSubcommand(cli.Subcmd("run", "
[OPTIONS] IMAGE [COMMAND] [ARG...]", "Run a command in a new container"), args,
nil)
```

其中，config的类型为Config结构体，hostConfig的类型为HostConfig结构体，两种类型的定义均位于runconfig包。Config与HostConfig类型同用以描述Docker容器的配置信息，然而两者之间又有着本质的区别，它们各自的描述如下：

·Config结构体：描述Docker容器独立的配置信息。独立的含义是：Config这部分信息描述的是容器本身，而不会与容器所在宿主机相关。

·HostConfig结构体：描述Docker容器与宿主机相关的配置信息。

1.Config结构体

Config结构体描述Docker容器本身的属性信息，而结构体内的所有属性正是说明了这一点，结构体的定义如下：

```
type Config struct {
     Hostname       string
     Domainname     string
     User           string
     Memory         int64  // Memory limit (in bytes)
     MemorySwap     int64  // Total memory usage (memory + swap); set '-1' to
disable
                          //swap
     CpuShares      int64  // CPU shares (relative weight vs. other containers)
     Cpuset         string // Cpuset 0-2, 0,1
     AttachStdin    bool
     AttachStdout   bool
     AttachStderr   bool
     PortSpecs      []string  // Deprecated - Can be in the format of 8080/tcp
     ExposedPorts   map[nat.Port]struct{}
     Tty             bool   // Attach standard streams to a tty, including stdin
if
                           // it is not closed.
     OpenStdin      bool   // Open stdin
     StdinOnce      bool   // If true, close stdin after the 1 attached client
                           // disconnects.
     Env            []string
     Cmd            []string
     Image          string // Name of the image as it was passed by the
                           // operator (eg. could be symbolic)
     Volumes        map[string]struct{}
     WorkingDir     string
     Entrypoint     []string
     NetworkDisabled bool
     OnBuild        []string
}
```

Config结构体中各属性的详细说明如表7-1所示。

表7-1　Config结构体属性介绍表

Config 结构体属性名	类型	代表含义
Hostname	string	容器主机名
Domainname	string	域名服务器名称
User	string	容器内用户名
Memory	int64	容器的内存使用上限（单位：字节）
MemorySwap	int64	容器所有的内存使用上限（物理内存 + 交互区），关闭交互区支持置为 -1
CpuShares	int64	容器 CPU 使用 share 值，其他容器的相对值
Cpuset	string	CPU 核的使用集合
AttachStdin	bool	是否附加标准输入
AttachStdout	bool	是否附加标准输出
AttachStderr	bool	是否附加标准出错
PortsSpecs	[]string	已弃用
ExposedPorts	map[nat.Port]struct{}	容器内部暴露的端口号
Tty	bool	是否分配一个伪终端 tty
OpenStdin	bool	在没有附加标准输入时，是否依然打开标准输入
StdinOnce	bool	若为真，表示当一个客户关闭标准输入后关闭容器的标准输入
Env	[]string	容器的环境变量，可以有多个
Cmd	[]string	容器内运行的指令（一个或多个）
Image	string	容器 rootfs 所依赖的镜像名称
Volumes	map[string]struct{}	容器从宿主机上挂载的目录
WorkingDir	string	容器内部进程的指定工作目录
Entrypoint	[]string	覆盖镜像中默认的 ENTRYPOINT
NetworkDisabled	bool	是否关闭容器网络功能
OnBuild	[]string	指定的命令在构建镜像时不执行，而是在镜像构建完成之后被触发执行

2.HostConfig结构体

HostConfig结构体描述Docker Container与宿主机相关的属性信息，结构体的定义如下：

```
type HostConfig struct {
     Binds              []string
     ContainerIDFile    string
     LxcConf            []utils.KeyValuePair
     Privileged         bool
     PortBindings       nat.PortMap
```

```
        Links               []string
        PublishAllPorts     bool
        Dns                 []string
        DnsSearch           []string
        VolumesFrom         []string
        Devices             []DeviceMapping
        NetworkMode         NetworkMode
        CapAdd              []string
        CapDrop             []string
        RestartPolicy       RestartPolicy
    }
```

Config结构体中各属性的详细说明如表7-2所示。

表7-2　HostConfig结构体属性介绍表

HostConfig 结构体属性名	类型	代表含义
Binds	[]string	从宿主机上绑定到容器的 volumes
ContainerIDFile	string	文件名，文件用以写入容器的 ID
LxcConf	[]utils.KeyValuePair	添加自定义的 lxc 选项键值对
Privileged	bool	是否将容器设置为特权模式
PortBindings	nat.PortMap	容器绑定到宿主机的端口
Links	[]string	与其他容器之间的 link 信息
PublishAllPorts	bool	是否在宿主机上映射容器所有的端口信息
Dns	[]string	自定义的 DNS 服务器地址
DnsSearch	[]string	自定义的 DNS 查找服务器地址
VolumesFrom	[]string	将自身 volumes 挂载到此容器的容器
Devices	[]DeviceMapping	为容器添加一个或多个宿主机设备
NetworkMode	NetworkMode	为容器设置的网络模式
CapAdd	[]string	为容器用户添加一个或多个 Linux Capabilities
CapDrop	[]string	为容器用户禁用一个或多个 Linux Capabilities
RestartPolicy	RestartPolicy	当一个容器退出时采取的重启策略

3.runconfig解析网络模式

分析完Config与HostConfig结构体之后，回到runconfig包中分析Docker Client如何解析与Docker容器网络模式相关的配置信息，

并将这部分信息传递给config实例与hostConfig实例。

runconfig包中的ParseSubcommand函数调用parseRun函数完成命令请求的分析，实现代码位于./docker/docker/runconfig/parse.go#L37-L39，如下所示：

```
func ParseSubcommand(cmd *flag.FlagSet, args []string, sysInfo *sysinfo.SysInfo)
(*Config, *HostConfig, *flag.FlagSet, error) {
    return parseRun(cmd, args, sysInfo)
}
```

进入parseRun函数即可发现，该函数完成了四方面的工作：

·定义与容器配置信息相关的flag参数。

·解析docker run命令后紧跟的请求内容，将请求内容全部保存至flag参数中。

·通过flag参数验证参数的有效性，并处理得到Config结构体与HostConfig结构体需要的属性值。

·创建并初始化Config类型实例config、HostConfig类型实例hostConfig，最终返回config、hostConfig与cmd。

本章分析Docker容器的网络模式，而parseRun函数中有关容器网络模式的flag参数有flNetwork与flNetMode，两者的定义分别位

于./docker/docker/runconfig/parse.go#L62

与./docker/runconfig/parse.go#L75，如下所示：

```
flNetwork = cmd.Bool([]string{"#n", "#-networking"}, true, "Enable networking for
this container")
flNetMode = cmd.String([]string{"-net"}, "bridge", "Set the Network mode for the
container\n'bridge': creates a new network stack for the container on the docker
bridge\n'none': no networking for this container\n'container:<name|id>': reuses
another container network stack\n'host': use the host network stack inside the
container.  Note: the host mode gives the container full access to local system
services such as D-bus and is therefore considered insecure.")
```

可见flag参数flNetwork表示是否开启容器的网络模式，若为true则开启，说明需要给容器创建网络环境；否则不开启，说明不给容器赋予网络功能。此flag参数的默认值为true，另外使用该flag的方式为：在docker run之后设定--networking或者-n，由于flNetwork的名称为{"#n"，"#-networking"}，故可以判定此flag已经处于弃用状态。

另一个flag参数flNetMode则表示用户为容器设定的网络模式，共有四种选项分别是：bridge、none、container：<name|id>和host。四种模式的作用上文已经有所介绍，此处不再赘述。使用此flag的方式为：在docker run之后设定--net，如：

```
docker run -it --net host IMAGE_NAME /bin/bash
```

用户使用docker run启动容器时设定了以上两个flag参数（实际只有flNetMode一个flag参数），则runconfig包会解析出这两个flag

的值。最终，通过flag参数flNetwork，得到Config类型实例config的属性NetworkDisabled；通过flag参数flNetMode，得到HostConfig类型实例hostConfig的属性NetworkMode。

函数parseRun返回config、hostConfig与cmd，代表着runconfig包解析配置参数工作的完成，下一节将回到CmdRun的运行。

7.3.3 CmdRun执行

在runconfig包中，Docker Client已经将有关容器网络模式的配置置于config对象与hostConfig对象，故在CmdRun函数的执行中，更多的是基于config对象与hostConfig参数的配置信息处理，而没有其他的容器网络处理部分。

CmdRun的主要工作是：利用Docker Daemon暴露的RESTful API接口，将docker run的请求发送至Docker Daemon。CmdRun执行过程中Docker Client与Docker Daemon的简易交互如图7-6所示。

从CmdRun的执行流程中，我们可以发现：在解析config、hostConfig与cmd之后，Docker Client首先发起请求create container。若Docker Daemon关于该容器的镜像存在，则立即执行create container操作并返回请求响应；Docker Client收到响应后，再发起请求start container。若容器镜像还不存在，Docker Daemon返回一个404的错误，表示镜像不存在；Docker Client收到错误响应之后，再发起一个请求pull image，使Docker Daemon首先下载镜像，下载完之后Docker Client再次发起请求create

container，Docker Daemon创建完之后，Docker Client最终发起请求start container。

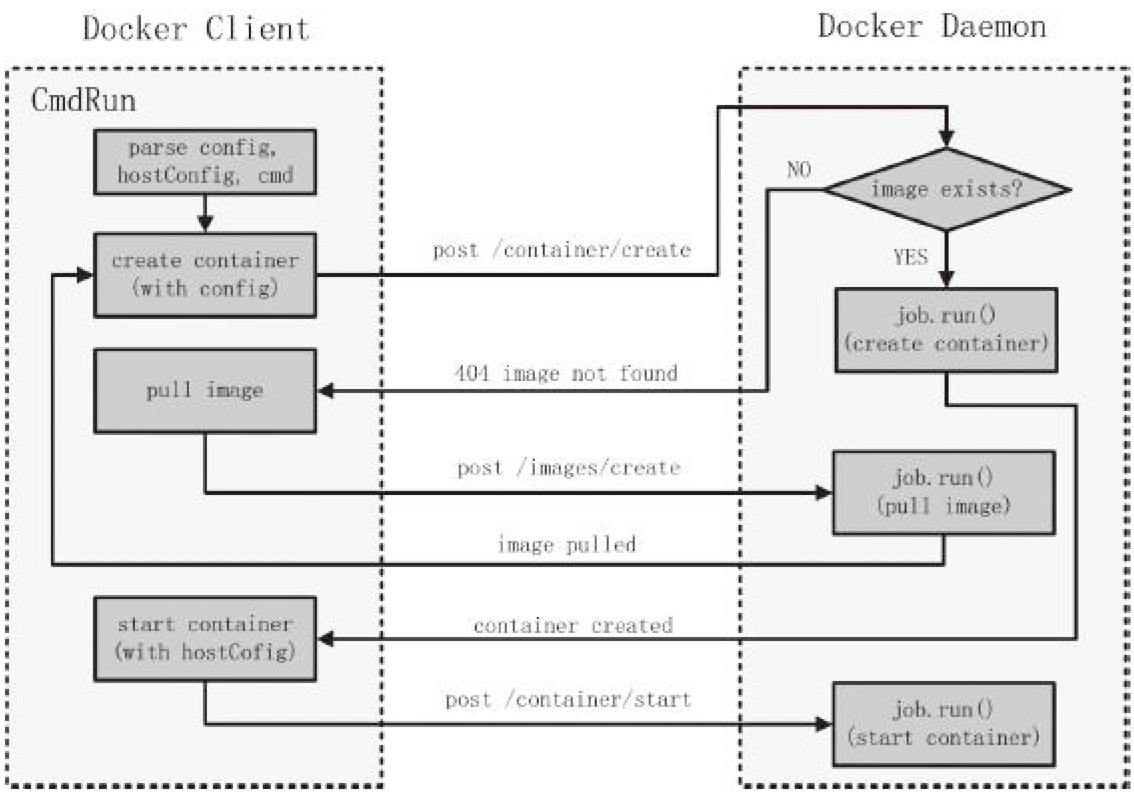


图7-6　Docker Client与Docker Daemon交互图

其中关于Docker容器网络模式的参数配置均存储于config与hostConfig对象之中，在请求create container和start container发起后，随请求一起发送至Docker Daemon。

7.4 Docker Daemon创建容器网络流程

Docker Daemon接收到Docker Client的请求可以分为两次，第一次为create container，第二次为start container。这两次请求的执行过程，都与Docker容器的网络相关。以下按照这两个请求的执行，具体分析Docker容器网络的创建。Docker Daemon如何通过Docker Server解析RESTful请求，并完成路由，在第5章已经详细分析过，故本章不再赘述。

7.4.1　创建容器之网络配置

Docker Daemon创建容器主要执行了create container操作。

创建容器过程中，Docker Daemon首先通过runconfig包中的ContainerConfigFromJob函数，解析出请求中的config对象，解析过程代码如下：

```
config := runconfig.ContainerConfigFromJob(job)
```

至此，Docker Client处理得到的config对象，已经传递至Docker Daemon的config对象，config对象中已经含有属性NetworkDisabled的具体值。

而容器创建的工作内容主要有以下两点：

1）创建与Docker容器对应的Container类型实例container。

2）创建Docker容器的rootfs。

具体的源码实现位于./docker/docker/daemon/create.go#L73-L78，如下所示：

```
if container, err = daemon.newContainer(name, config, img); err != nil {
     return nil, nil, err
}
if err := daemon.createRootfs(container, img); err != nil {
```

```
        return nil, nil, err
    }
```

与Docker容器网络模式配置相关的内容主要位于第一点，即创建container实例中。newContainer函数的定义位于./docker/docker/daemon/daemon.go#L516-L550，具体的container实例如下：

```
container := &Container{
        ID:              id,
        Created:         time.Now().UTC(),
        Path:            entrypoint,
        Args:            args, //FIXME: de-duplicate from config
        Config:           config,
        hostConfig:       &runconfig.HostConfig{},
        Image:           img.ID, // Always use the resolved image id
        NetworkSettings: &NetworkSettings{},
        Name:            name,
        Driver:          daemon.driver.String(),
        ExecDriver:      daemon.execDriver.Name(),
        State:           NewState(),
}
```

在container对象中，config对象直接赋值给container对象的Config属性，另外hostConfig属性与NetworkSettings属性均为空。其中hostConfig对象将在start container请求执行过程中被赋值，NetworkSettings类型的作用是描述容器的网络具体信息，定义位于./docker/docker/daemon/network_settings.go#L11-L18，源码如下所示：

```
type NetworkSettings struct {
        IPAddress   string
        IPPrefixLen int
        Gateway     string
        Bridge      string
        PortMapping map[string]PortMapping // Deprecated
```

```
    Ports       nat.PortMap
}
```

Networksettings类型的各属性详细说明如表7-3所示。

表7-3　NetworkSettings属性介绍表

NetworkSettings 属性名称	类型	含义
IPAddress	string	容器网络接口的 IP 网络地址
IPPrefixLen	int	网络标识位长度
Gateway	string	容器的默认网关地址
Bridge	string	容器网络接口使用的网桥地址
PortMapping	map[string]PortMapping	容器与宿主机的端口映射
Ports	nat.PortMap	容器内部暴露的端口号

7.4.2　启动容器之网络配置

创建容器阶段，Docker Daemon创建了容器对象container，container对象内部的Config属性含有NetworkDisabled。创建容器完成之后，Docker Daemon应Docker Client的请求，需要执行create container的操作。这部分的执行同样需要Docker Server接收请求，并分发调度处理。

Docker Daemon启动容器主要执行了start container操作。

启动容器过程中，Docker Daemon首先通过runconfig包中的ContainerHostConfigFromJob函数，解析出请求中的hostConfig对象，解析过程源码如下：

```
hostConfig := runconfig.ContainerHostConfigFromJob(job)
```

至此，Docker Client处理得到的hostConfig对象，已经传递至Docker Daemon的hostConfig对象，hostConfig对象中已经含有属性NetworkMode具体值。

容器启动的所有工作，均由以下Start函数来完成，源码位于./docker/docker/daemon/start.go#L36-L38，如下所示：

```
if err := container.Start(); err != nil {
     return job.Errorf("Cannot start container %s: %s", name, err)
}
```

Start函数实现了容器的启动。更为具体的描述是：Start函数实现了进程的启动，另外在启动进程的同时为进程设定了命名空间（namespaces）并为进程做了资源的限制，从而保证进程以及之后进程的子进程都会在相同的命名空间内，且受到相同的资源控制。如此一来，Start函数创建的进程，以及该进程之后的子进程，形成一个进程组，该进程组处于资源隔离和资源控制的环境中，我们习惯将这样的进程组环境称为容器，也就是这里的Docker容器。

回到Start函数的执行，位于./docker/docker/daemon/container.go#L275-L320。Start函数的执行过程与Docker容器网络模式相关的主要有三部分：

·initializeNetwork，初始化container对象中与网络相关的属性。

·populateCommand，填充Docker容器内部需要执行的命令，Command中含有进程启动命令，还含有容器环境的配置信息，也包括网络配置。

·container.waitForStart（），实现Docker容器内部进程的启动，进程启动之后，为容器创建网络环境等。

1.初始化容器网络配置

容器对象container中有属性hostConfig，属性hostConfig中有属性NetworkMode，初始化容器网络配置initializeNetworking（）的主要工作就是：通过NetworkMode属性为Docker容器的网络做相应的初始化配置工作。

Docker Container的网络模式有四种，分别为：host、other container、none以及bridge。initializeNetworking函数的执行完全覆盖了这四种模式。

initializeNetworking（）函数的源码实现位于./docker/docker/daemon/container.go#L881-L933，如下所示：

```
func (container *Container) initializeNetworking() error {
var err error
if container.hostConfig.NetworkMode.IsHost() {
     container.Config.Hostname, err = os.Hostname()
     if err != nil {
          return err
     }
     parts := strings.SplitN(container.Config.Hostname, ".", 2)
     if len(parts) > 1 {
          container.Config.Hostname = parts[0]
          container.Config.Domainname = parts[1]
     }
     content, err := ioutil.ReadFile("/etc/hosts")
     if os.IsNotExist(err) {
           return container.buildHostnameAndHostsFiles("")
     } else if err != nil {
           return err
     }
     if err := container.buildHostnameFile(); err != nil {
           return err
     }
     hostsPath, err := container.getRootResourcePath("hosts")
     if err != nil {
           return err
```

```
        }
        container.HostsPath = hostsPath
        return ioutil.WriteFile(container.HostsPath, content, 0644)
    } else if container.hostConfig.NetworkMode.IsContainer() {
        // we need to get the hosts files from the container to join
        nc, err := container.getNetworkedContainer()
        if err != nil {
              return err
        }
        container.HostsPath = nc.HostsPath
        container.ResolvConfPath = nc.ResolvConfPath
        container.Config.Hostname = nc.Config.Hostname
        container.Config.Domainname = nc.Config.Domainname
    } else if container.daemon.config.DisableNetwork {
        container.Config.NetworkDisabled = true
        return container.buildHostnameAndHostsFiles("127.0.1.1")
    } else {
        if err := container.allocateNetwork(); err != nil {
              return err
        }
        return
    container.buildHostnameAndHostsFiles(container.NetworkSettings.IPAddress)
    }
    return nil
    }
```

针对以上源码，下面以4种不同的网络模式分析initializeNetworking函数的作用。

第一，host网络模式。Docker容器网络的host模式意味着容器使用宿主机的网络环境。虽然Docker容器使用宿主机的网络环境，但这并不代表Docker容器可以拥有宿主机文件系统的视角，而host宿主机上有很多信息标识的是网络信息，故Docker Daemon需要将这部分标识网络的信息，从宿主机上添加到Docker容器内部的指定位置。这样的网络信息，主要有以下三种：

·宿主机的hostname文件，代表容器的主机名。

·宿主机的hosts文件，代表容器的主机名配置文件。

·宿主机resolv.conf文件，属于容器的域名文件。

由于Docker容器与宿主机共享网络，因此Docker容器的hostname文件、hosts文件以及resolv.conf文件原则上与宿主机上的这些文件内容应该保持一致。结果就是：Docker Daemon将宿主机的hostname文件、hosts文件以及resolv.conf内容写入Docker容器的指定目录下。目录一般为/var/lib/docker/containers/<container_id>，当容器启动时，再将这部分文件挂载进容器内部的指定路径。

第二，other container网络模式。Docker容器的other container网络模式意味着：容器使用其他已经创建容器的网络环境。

Docker Daemon首先判断host网络模式，若不为host网络模式，则继续判断是否为other container模式。若Docker容器的网络模式为other container（假设使用的-net参数为--net=container：17adef，其中17adef为容器ID），则用户必须指定被共享网络的Docker容器，此处容器ID为17adef。在这种情况下，Docker Daemon所做的执行操作包括两步。

第一步，从container对象的hostConfig属性中找出NetworkMode，并找到相应的容器，即17adef的容器对象container，实现源码如下：

```
nc, err := container.getNetworkedContainer()
```

第二步，将17adef容器对象的HostsPath、ResolveConfPath、Hostname和Domainname赋值给当前容器对象container，实现源码如下：

```
container.HostsPath = nc.HostsPath
container.ResolvConfPath = nc.ResolvConfPath
container.Config.Hostname = nc.Config.Hostname
container.Config.Domainname = nc.Config.Domainname
```

第三，none网络模式。Docker容器的none网络模式意味着不给该容器创建任何网络环境，容器只能使用127.0.1.1的环回接口。

Docker Daemon通过config属性的DisableNetwork来判断是否为none网络模式。实现源码如下：

```
if container.daemon.config.DisableNetwork {
     container.Config.NetworkDisabled = true
     return container.buildHostnameAndHostsFiles("127.0.1.1")
}
```

第四，bridge网络模式。Docker容器的bridge网络模式意味着为容器创建桥接网络模式。桥接模式使得Docker容器创建独立的网络环境，并通过“桥接”的方式实现Docker容器与外界的网络通信。

初始化bridge网络模式的配置，实现源码如下：

```
if err := container.allocateNetwork(); err != nil {
      return err
```

```
}
return container.buildHostnameAndHostsFiles(container.NetworkSettings.IPAddress)
```

以上代码完成的内容主要也是两部分：第一，通过allocateNetwork函数为容器分配一个网络接口可用的IP地址，并将网络配置（包括IP、bridge、Gateway等）赋值给container对象的NetworkSettings；第二，通过NetworkSettings为容器创建hosts、hostname等文件。

2.创建容器Command信息

Docker在实现容器时，在源码层次使用了Command类型。Command是一个非常重要的概念，我们可以认为Command类型包含了两部分的内容：第一，运行容器内进程的外部命令exec.Cmd；第二，运行容器时启动进程需要的所有基础信息：包括容器进程组的使用资源、网络资源、使用设备、工作路径等。通过这两部分的内容，我们清楚如何创建容器内的进程，同时也清楚为容器创建什么样的环境。

首先，我们先来看Command类型的定义，位于./docker/docker/daemon/execdriver/driver.go#L84。通过分析Command类型以及相关的其他数据结构类型，得到的Command类型关系如图7-7所示。

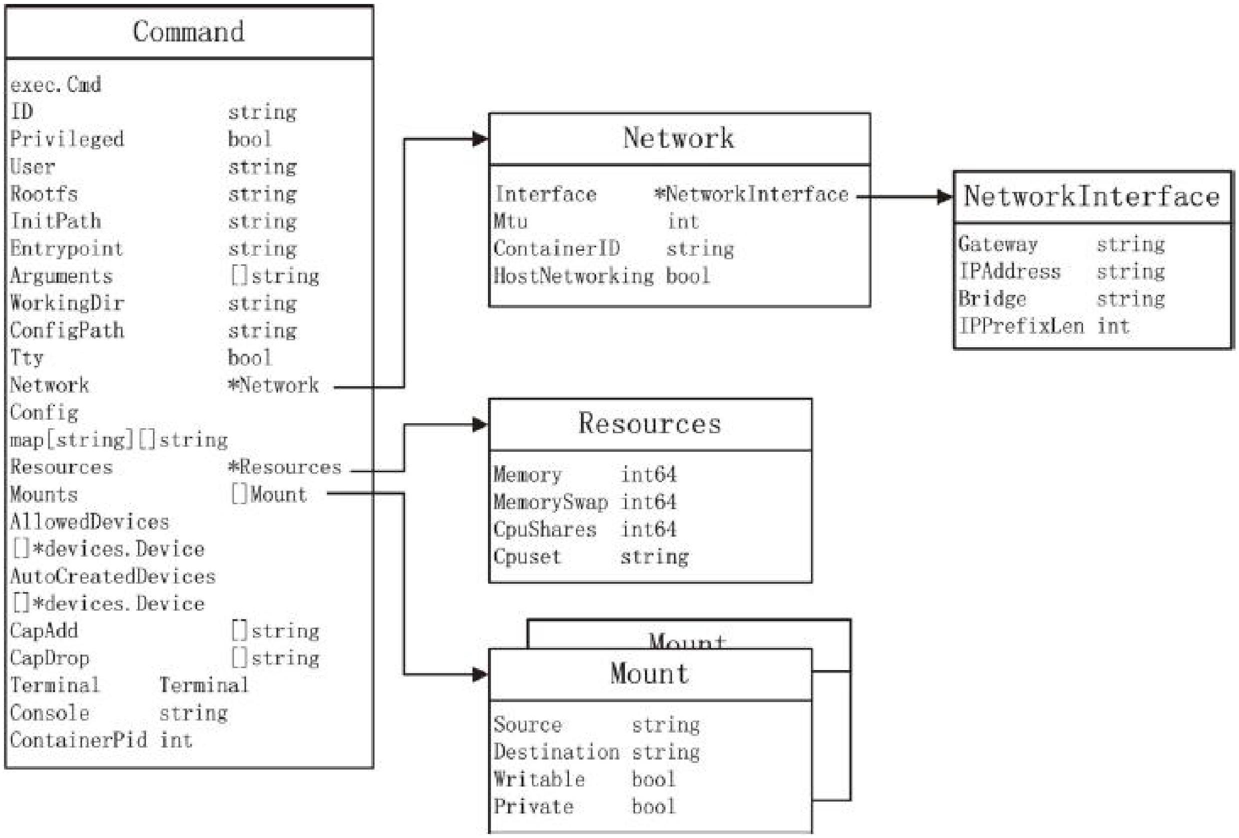


图7-7　Command类型关系图

从Command类型关系图中，我们可以看到Command类型中第一个属性为exec.Cmd，即代表需要创建的进程具体的外部命令；同时，关于网络方面的属性有Network，Network的类型为*Network；关于Docker容器资源使用方面的属性为Resources，从Resource的类型来看，Docker目前能做的资源限制有4个维度，分别为内存，内存+Swap，CPU使用，CPU核使用；关于挂载的内容，有属性Mounts；等等。

简单介绍Command类型之后，回到Docker Daemon启动容器网络的源码分析。紧接在函数initializeNetworking之后，是

populateCommand环节。populateCommand的函数实现位于./docker/docker/daemon/container.go#L191-L274。上文已经提及，populateCommand的作用是创建execdriver包中的对象Command实例，该Command中既有启动容器进程的外部命令，同时也有众多容器环境的配置信息，包括网络。

本小节，分析populateCommand如何填充Command对象中的网络信息，其他内容的分析会在第12章和第13章进行展开。

Docker容器有四种网络模式，故populateCommand首先需要判断容器属于哪种网络模式，随后将具体的网络模式信息，写入Command对象的Network属性中。查验Docker容器网络模式的源码位于./docker/docker/daemon/container.go#L204-L227，如下所示：

```
parts := strings.SplitN(string(c.hostConfig.NetworkMode), ":", 2)
switch parts[0] {
case "none":
case "host":
     en.HostNetworking = true
case "bridge", "": // empty string to support existing containers
     if !c.Config.NetworkDisabled {
           network := c.NetworkSettings
           en.Interface = &execdriver.NetworkInterface{
                 Gateway:     network.Gateway,
                 Bridge:      network.Bridge,
                 IPAddress:   network.IPAddress,
                 IPPrefixLen: network.IPPrefixLen,
          }
     }
case "container":
     nc, err := c.getNetworkedContainer()
     if err != nil {
          return err
     }
     en.ContainerID = nc.ID
default:
```

```
        return fmt.Errorf("invalid network mode: %s", c.hostConfig.NetworkMode)
    }
```

populateCommand首先通过hostConfig对象中的NetworkMode判断容器属于哪种网络模式。该部分内容涉及execdriver包中的Network，可参见Command类型关系图中的Network类型。若为none模式，则对于Network对象（即en，*execdriver.Network）不做任何操作。若为host模式，则将Network对象的HostNetworking置为true；若为bridge桥接模式，则首先创建一个NetworkInterface对象，完善该对象的Gateway、Bridge、IPAddress和IPPrefixLen信息，最后将NetworkInterface对象作为Network对象的中Interface属性的值；若为other container模式，则首先通过getNetworkedContainer（）函数获知被分享网络命名空间的容器，最后将容器ID，赋值给Network对象的ContainerID。由于bridge模式、host模式以及other container模式彼此互斥，故Network对象中Interface属性、ContainerID属性以及HostNetworking三者之中只有一个被赋值。当Docker容器的网络被查验之后，populateCommand将en实例Network属性的值，传递给Command对象。

3.启动容器内部进程

当为容器做好所有的配置之后，Docker Daemon需要真正意义上的启动容器。根据启动容器流程中涉及的Docker模块，Docker

Daemon后续环节中实现启动容器的请求会被发送至execdriver，再经过libcontainer，最后实现Linux内核级别的进程启动。

回到Docker Daemon的启动容器，daemon包中start函数的最后一步即为执行container.waitForStart（）。waitForStart函数的定义位于./docker/daemon/container.go#L1070-L1082，源码如下：

```
func (container *Container) waitForStart() error {
	container.monitor = newContainerMonitor(container,
container.hostConfig.RestartPolicy)
	select {
	case <-container.monitor.startSignal:
	case err := <-utils.Go(container.monitor.Start):
		return err
	}
	return nil
}
```

以上源码运行过程中，Docker Daemon首先通过函数newContainerMonitor返回一个初始化的containerMonitor对象，该对象中带有容器进程的重启策略。总体而言，containerMonitor对象用以监视容器进程的执行。容器进程指的是容器pid namespace内进程号为1的进程，这个进程的状态代表容器的状态。一旦该进程停止运行，则容器内部所有进程都将收到一个终止信号，最终导致容器的运行失败。如果containerMonitor中指定了进程的重启策略，那么一旦容器进程没有启动成功，Docker Daemon会使用重启策略来重启容器。如果在重启策略下，容器依然没有成功启动，那么containerMonitor对象会负责重置以及清除所有已经为容器准备好的

资源，例如已经为容器分配好的网络资源（即IP地址），还有为容器准备的rootfs等。

waitForStart（）函数通过container.monitor.Start来实现容器的启动，进入./docker/docker/daemon/monitor.go#L100，可以发现启动容器进程位于./docker/docker/daemon/monitor.go#L136，源码如下：

```
exitStatus, err = m.container.daemon.Run(m.container, pipes, m.callback)
```

以上源码实际调用了daemon包中的Run函数，位于./docker/daemon/daemon.go#L969-L971，如下所示：

```
func (daemon *Daemon) Run(c *Container, pipes *execdriver.Pipes, startCallback
execdriver.StartCallback) (int, error) {
     return daemon.execDriver.Run(c.command, pipes, startCallback)
}
```

最终，Run函数中调用了execdriver中的Run函数来执行Docker Container的启动命令。

至此，网络部分在Docker Daemon内部的执行已经结束，紧接着程序的运行陷入execdriver，进一步运行容器启动的相关步骤。

7.5 execdriver网络执行流程

Docker架构中execdriver的作用是启动容器内部进程，最终启动容器。目前，在Docker中execdriver作为执行驱动，可以有两种选项：lxc与native。其中，lxc驱动会调用lxc工具实现容器的启动，而native驱动会使用Docker官方发布的libcontainer来启动容器。

Docker Daemon启动过程中，execdriver的类型默认为native，故本章主要分析native驱动在执行启动容器时，如何处理网络部分。

在Docker Daemon启动容器的最后一步，即调用了execdriver的Run函数来执行。通过分析Run函数的具体实现，可知关于Docker容器的网络执行流程主要包括两个环节：

1）创建libcontainer的Config对象。

2）通过libcontainer中的namespaces包执行启动容器。

将execdriver.Run函数的运行流程展开，与Docker容器网络相关的流程，如图7-8所示。

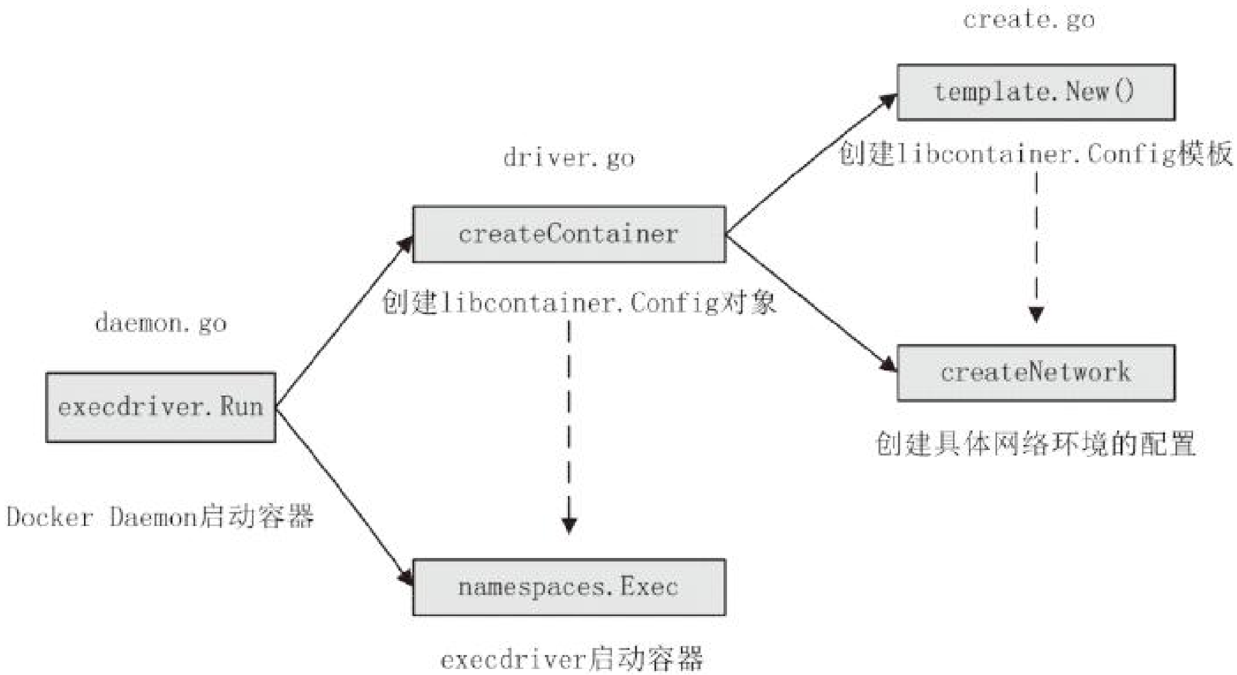


图7-8　execdriver.Run执行流程图

7.5.1　创建libcontainer的Config对象

Run函数位于./docker/docker/daemon/execdriver/native/driver.go#L62-L168。进入Run函数的实现，立即可以发现该函数通过createContainer创建了一个container对象，源码如下：

```
container, err := d.createContainer(c)
```

其中c为Docker Daemon创建的execdriver.Command类型实例。以上源码中createContainer函数的作用是：使用execdriver.Command来填充libcontainer.Config。简要介绍libcontainer.Config的作用就是，它定义了在一个容器化的环境中执行一个进程所需要的所有配置项。createContainer函数使用Docker Daemon层创建的execdriver.Command，创建更底层libcontainer所需要的Config对象。从这个角度来看，execdriver更像是封装了libcontainer对外的接口，实现了将Docker Daemon特有的容器启动信息转换为底层libcontainer能真正使用的容器配置选项。libcontainer.Config类型与其内部对象之间的关系如图7-9所示。

createContainer的源码实现部分位于./docker/docker/daemon/execdriver/native/create.go#L23-

L77，如下所示：

```
func (d *driver) createContainer(c *execdriver.Command) (*libcontainer.Config, 
error) {
     container := template.New()
     ...
     if err := d.createNetwork(container, c); err != nil {
          return nil, err
     }
     ...
     return container, nil
}
```

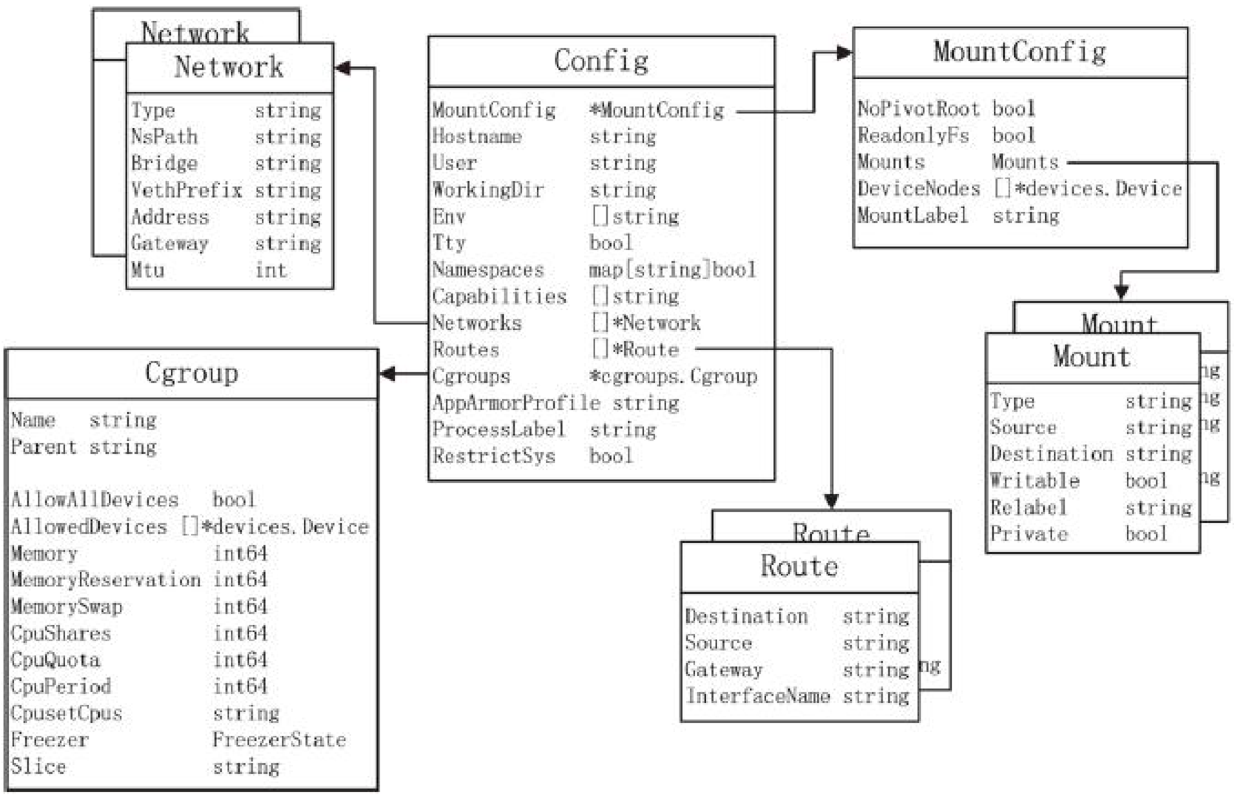


图7-9　libcontainer.Config类型关系图

1.libcontainer.Config模板实例

从createContainer函数的实现以及execdriver.Run执行流程图中都可以看到，create-Container所做的第一个操作就是执行

template.New（ ），意为创建一个libcontainer.Config的实例容器。其中，template.New（ ）的定义位于./docker/docker/daemon/execdriver/native/template/default_template.go，主要的作用为返回libcontainer关于Docker容器的默认配置选项。

Template.New（ ）的源码实现如下：

```
func New() *libcontainer.Config {
     container := &libcontainer.Config{
          Capabilities: []string{
               "CHOWN",
               "DAC_OVERRIDE",
               "FSETID",
               "FOWNER",
               "MKNOD",
               "NET_RAW",
               "SETGID",
               "SETUID",
               "SETFCAP",
               "SETPCAP",
               "NET_BIND_SERVICE",
               "SYS_CHROOT",
               "KILL",
               "AUDIT_WRITE",
          },
          Namespaces: map[string]bool{
               "NEWNS":  true,
               "NEWUTS": true,
               "NEWIPC": true,
               "NEWPID": true,
               "NEWNET": true,
          },
          Cgroups: &cgroups.Cgroup{
               Parent:          "docker",
               AllowAllDevices: false,
          },
          MountConfig: &libcontainer.MountConfig{},
     }
     if apparmor.IsEnabled() {
          container.AppArmorProfile = "docker-default"
     }
     return container
}
```

libcontainer.Config默认的模板对象，首先设定了Capabilities的默认项，如CHOWN、DAC_OVERRIDE、FSETID等；其次又为Docker容器所需设定的namespaces添加默认值，即需要创建5个命名空间，如NEWNS、NEWUTS、NEWIPC、NEWPID和NEWNET，其中不包括user namespace，另外与网络相关的namespace是NEWNET；最后设定了一些关于cgroup以及apparmor的默认配置。

Template.New（ ）函数最后返回类型为libcontainer.Config的实例container，该实例中只包含默认配置项，其他内容的添加需要createContainer的后续代码来完成。

2.createNetwork实现

在createContainer的实现流程中，为了完善container对象（类型为libcontainer.Config），后续还有很多步骤，如与网络相关的createNetwork函数调用，与Linux内核Capabilities相关的setCapabilities函数调用，与cgroups相关的setupCgroups函数调用，以及与挂载目录相关的setupMounts函数调用等。本小节主要分析createNetwork函数如何为container对象完善网络配置项。

createNetwork函数的定义位于./docker/docker/daemon/execdriver/native/create.go#L79-L124，该函数主要利用execdriver.Command中Network属性的内

容，来判断如何创建libcontainer.Config中的Network属性（关于两种Network属性，可以参见图7-7和图7-9）。由于Docker容器的4种网络模式彼此互斥，故以上Network类型中Interface、ContainerID与HostNetworking最多只有一项会被赋值。

execdriver.Command中Network的类型定义如下：

```
type Network struct {
     Interface      *NetworkInterface
     Mtu            int
     ContainerID    string
     HostNetworking bool
}
```

在以上Network类型的基础上，我们分析createNetwork函数，其具体实现可以归纳为以下4个部分：

1）判断网络是否为host模式。

2）判断网络是否为bridge桥接模式。

3）判断网络是否为other container模式。

4）为Docker容器添加loopback网络设备。

首先，我们来看execdriver判断容器网络是否为host模式的源码：

```
if c.Network.HostNetworking {
     container.Namespaces["NEWNET"] = false
```

```
		return nil
	}
```

当execdriver.Command类型实例中Network属性的HostNetworking为true，则说明需要为Docker容器创建host网络模式，使容器与宿主机共享相同的网络命名空间。在host模式的具体介绍中，我们已经阐明，只要Docker Daemon在创建容器进程，进行CLONE系统调用时，不传入CLONE_NEWNET参数标志即可实现共享网络命名空间。这部分源码正好准确地验证了这一点。Docker Daemon将container对象中代表网络命名空间的NEWNET设为false，最终导致libcontainer中不创建新的网络命名空间。

再来看，execdriver判断容器网络是否为bridge桥接模式的源码：

```
	if c.Network.Interface != nil {
		vethNetwork := libcontainer.Network{
			Mtu:        c.Network.Mtu,
				Address:    fmt.Sprintf("%s/%d", c.Network.Interface.IPAddress,
	c.Network.Interface.IPPrefixLen),
			Gateway:    c.Network.Interface.Gateway,
			Type:       "veth",
			Bridge:     c.Network.Interface.Bridge,
			VethPrefix: "veth",
		}
		container.Networks = append(container.Networks, &vethNetwork)
	}
```

当execdriver.Command类型实例中Network属性的Interface不为nil值，则说明需要为Docker容器创建bridge桥接模式，使容器拥有隔离的网络环境。于是，以上源码为libcontainer.Config的

container对象添加Networks属性vethNetwork，网络接口类型为veth，以便libcontainer在执行时，可以为Docker容器创建veth pair。

接着来看execdriver判断容器网络是否为other container模式的代码：

```
if c.Network.ContainerID != "" {
     d.Lock()
     active := d.activeContainers[c.Network.ContainerID]
     d.Unlock()
     if active == nil ||  active.cmd.Process == nil {
               return fmt.Errorf("%s is not a valid running container to join",
c.Network.ContainerID)
     }
     cmd := active.cmd
     nspath := filepath.Join("/proc", fmt.Sprint(cmd.Process.Pid), "ns", "net")
     container.Networks = append(container.Networks, &libcontainer.Network{
          Type:   "netns",
          NsPath: nspath,
     })
}
```

当execdriver.Command类型实例中Network属性的ContainerID不为空字符串时，则说明需要为Docker容器创建other container模式，使创建容器共享其他容器的网络环境。实现过程中，execdriver首先需要在activeContainers中查找需要被共享网络环境的容器active；并通过active容器的启动执行命令cmd找到容器主进程在宿主机上的PID；随后在proc文件系统中找到该进程PID的关于网络命名空间的路径nspath，也是整个容器的网络命名空间路径；最后为类型为libcontainer.Config的container对象添加Networks属性，Network的类型为netns。

此外，createNetwork函数还实现了为Docker容器创建一个loopback环回接口，以便容器可以实现内部通信。实现过程中，同样为类型libcontainer.Config的container对象添加Networks属性，Network的类型为loopback，源码如下：

```
container.Networks = []*libcontainer.Network{
     {
          Mtu:      c.Network.Mtu,
          Address: fmt.Sprintf("%s/%d", "127.0.0.1", 0),
          Gateway: "localhost",
          Type:     "loopback",
     },
}
```

至此，createNetwork函数已经把与网络相关的配置，全部创建在类型为libcontainer.Config的container对象中，随时等候创建容器进程的来临。

7.5.2　调用libcontainer的namespaces启动容器

回到execdriver.Run函数，创建完libcontainer.Config实例container，经过一系列处理之后，最终execdriver执行namespaces.Exec函数实现启动容器。启动过程中container对象依然是namespace.Exec函数中一个非常重要的参数。namespaces.Exec代表着execdriver把启动Docker容器的工作交给libcontainer，之后的程序执行将完全陷入libcontainer。

调用namespaces.Exec的源码位于./docker/daemon/execdriver/native/driver.go#L102-L127，为了便于理解，简化之后如下：

```
namespaces.Exec(container, c.Stdin, c.Stdout, c.Stderr, c.Console, c.Rootfs,
dataPath, args, parameter_1, parameter_2)
```

其中parameter_1为定义的函数，如下所示：

```
func(container *libcontainer.Config, console, rootfs, dataPath, init string,
child *os.File, args []string) *exec.Cmd {
     c.Path = d.initPath
     c.Args = append([]string{
          DriverName,
          "-console", console,
          "-pipe", "3",
          "-root", filepath.Join(d.root, c.ID),
          "--",
```

```
	}, args...)
	// set this to nil so that when we set the clone flags anything else is
reset
	c.SysProcAttr = &syscall.SysProcAttr{
		Cloneflags:
uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
	}
	c.ExtraFiles = []*os.File{child}
	c.Env = container.Env
	c.Dir = c.Rootfs
	return &c.Cmd
}
```

同样，parameter_2也为定义的函数，如下所示：

```
func() {
	if startCallback != nil {
		c.ContainerPid = c.Process.Pid
		startCallback(c)
	}
}
```

parameter_1以及parameter_2这两个函数均会在libcontainer的namespaces中发挥很大的作用，7.6节将进行深入分析。

至此，execdriver模块的执行部分已经结束，Docker Daemon的运行陷入libcontainer。

7.6 libcontainer实现内核态网络配置

libcontainer是一个Linux操作系统上容器技术的解决方案。libcontainer指定了创建一个容器时所需要的配置选项，同时它利用Linux中namespaces和cgroups等技术为使用者提供了一套Golang原生态的容器实现方案，并且没有使用任何外部依赖。用户借助libcontainer，可以感受到众多操作命名空间、网络等资源的便利。

当execdriver调用libcontainer中namespaces包的Exec函数时，libcontainer开始发挥其实现容器功能的作用。Exec函数位于./docker/libcontainer/namespaces/exec.go#L24-L113。本节更多地关心Docker容器的网络创建，因此从这个角度来看Exec的实现可以分为三个步骤：

1）通过createCommand创建一个Golang语言内的exec.Cmd对象。

2）启动命令exec.Cmd，创建容器内的第一个进程。

3）通过InitializeNetworking函数为容器进程初始化网络环境。

以下从源码实现的角度，详细分析这三个部分。

7.6.1　创建exec.Cmd

提到exec.Cmd，就不得不提Go语言标准库中的os包和os/exec包。前者提供了与平台无关的操作系统功能集，后者则提供了功能集里与命令执行相关的部分。

首先来看一下在Go语言中exec.Cmd的定义，如下所示：

```
type Cmd struct {
     Path string                           //所需执行命令在系统中的路径
     Args []string                         //传入命令的参数
     Env []string                          //进程运行时的环境变量
     Dir string                            //命令运行的工作目录
     Stdin io.Reader
     Stdout io.Writer
     Stderr io.Writer
     ExtraFiles []*os.File                 //进程所需打开的额外文件
     SysProcAttr *syscall.SysProcAttr      //可选的操作系统属性
     Process *os.Process                   //代表Cmd启动后，操作系统底层的具体进程
     ProcessState *os.ProcessState         //进程退出后保留的信息
}
```

清楚Cmd的定义之后，再来分析namespaces包中的Exec函数，libcontainer是如何来创建exec.Cmd的。在Exec函数的实现过程中，使用了以下源码实现Exec.Cmd的创建：

```
command := createCommand(container, console, rootfs, dataPath, os.Args[0],
syncPipe.Child(), args)
```

其中createCommand为namespace.Exec函数中传入的倒数第二个参数，类型为CreateCommand。而createCommand只是

namespaces.Exec函数的形参，真正的实参则为execdriver调用namespaces.Exec时的参数parameter_1，源码如下：

```
func(container *libcontainer.Config, console, rootfs, dataPath, init string,
child *os.File, args []string) *exec.Cmd {
     c.Path = d.initPath
     c.Args = append([]string{
          DriverName,
          "-console", console,
          "-pipe", "3",
          "-root", filepath.Join(d.root, c.ID),
          "--",
     }, args...)
     // set this to nil so that when we set the clone flags anything else is
reset
     c.SysProcAttr = &syscall.SysProcAttr{
          Cloneflags:
uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
     }
     c.ExtraFiles = []*os.File{child}
     c.Env = container.Env
     c.Dir = c.Rootfs
     return &c.Cmd
}
```

熟悉exec.Cmd的定义之后，分析以上源码就显得较为简单。为Cmd赋值的对象有Path、Args、SysProcAttr、ExtraFiles、Env和Dir。其中需要特别注意的是Path的值d.initPath，该路径下存放的是dockerinit的二进制文件，Docker 1.2.0版本下，路径一般为“/var/lib/docker/init/dockerinit-1.2.0”。另外SysProcAttr使用以下的代码来赋值：

```
&syscall.SysProcAttr{
     Cloneflags: uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
}
```

在syscall.SysProAttr对象中的Cloneflags属性中，即保留了libcontainer.Config类型的实例container中的Namespaces属性。换言之，通过exec.Cmd创建进程时，libcontainer正是通过Cloneflags来实现在Clone系统调用中传入namespaces参数标志。

回到函数执行中，在函数的最后返回了c.Cmd，命令创建完毕。

7.6.2　启动exec.Cmd创建进程

创建完exec.Cmd，当然需要执行该命令，namespaces.Exec函数中直接使用以下源码实现进程的启动：

```
if err := command.Start(); err != nil {
     return -1, err
}
```

这一部分的内容简单直接，Start（ ）函数用以完成指定命令exec.Cmd的启动执行，同时不等待其启动完便返回。Start（ ）函数的定义位于os/exec包。

进入os/exec包，查看Start（ ）函数的实现，可以看到执行过程中，会对command.Process进行赋值，此时command.Process中会含有刚才启动进程的PID进程号，该Pid号在宿主机pid namespace下，而并非是新创建的namespace下的Pid号。

7.6.3　为容器进程初始化网络环境

上一环节实现了容器进程的启动，然而还没有为之配置相应的网络环境。namespaces.Exec在之后的InitializeNetworing中实现了为容器进程初始化网络环境。初始化网络环境需要两个非常重要的参数：container对象以及容器进程的Pid号。类型为libcontainer.Config的实例container包含用户对Docker容器的网络配置需求，另外容器进程的Pid可以使得创建的网络环境与进程新创建的namespace进行关联。

namespaces.Exec中为容器进程初始化网络环境的代码实现位于./libcontainer/namespaces/exec.go#L75-L79，如下所示：

```
if err := InitializeNetworing(container, command.Process.Pid, syncPipe,
&networkState); err != nil {
     command.Process.Kill()
     command.Wait()
     return -1, err
}
```

InitializeNetworing的作用很明显，即为创建的容器进程初始化网络环境。具体实现包含两个步骤：

1）先在容器进程的网络命名空间外部创建该容器所需的网络栈。

2）将创建的网络栈传递至容器的网络命名空间。

IntializeNetworking的源代码实现位于./libcontainer/namespaces/exec.go#L176-L187，如下所示：

```
func InitializeNetworking(container *libcontainer.Config, nspid int, pipe
*syncpipe.SyncPipe, networkState *network.NetworkState) error {
     for _, config := range container.Networks {
          strategy, err := network.GetStrategy(config.Type)
          if err != nil {
               return err
          }
               if err := strategy.Create((*network.Network)(config), nspid,
networkState); err != nil {
               return err
          }
     }
     return pipe.SendToChild(networkState)
}
```

以上源码实现过程，首先通过一个循环，遍历libcontainer.Config类型实例container中的网络属性Networks；随后使用GetStrategy函数处理Networks中每一个对象的Type属性，得出Network的类型，这里的类型有3种，分别为loopback、veth和netns。loopback环回接口针对bridge模式和none模式；veth针对bridge模式；而netns针对other container模式。

得到Network的类型之后，libcontainer创建相应的网络栈，具体实现使用每种网络栈类型下的Create函数。下面分析三种不同网络栈各自的创建流程。需要额外注意的是：容器网络栈的创建不在容器

网络命名空间之内，而是在Docker Daemon所在的网络命名空间内。

1.loopback网络栈的创建

loopback是一种本地环回接口，libcontainer创建loopback网络设备的实现代码位于./libcontainer/network/loopback.go#L13-L15，如下所示：

```
func (l *Loopback) Create(n *Network, nspid int, networkState *NetworkState)
error {
	return nil
}
```

令人费解的是，libcontainer在loopback接口的创建函数Create中，并没有实质性的内容，而是直接返回nil。其实关于loopback接口的创建，要回到Linux内核为进程新建net namespace的阶段。当libcontainer执行command.Start（ ）时，由于创建了一个新的net namespace，故Linux内核会自动为新的net namespace创建一个loopback接口。当Linux内核创建完loopback接口之后，libcontainer所做的工作即只保留loopback设备的默认配置，并在后续libcontainer的Initialize函数中实现启动该接口。

2.veth网络栈的创建

veth是Docker容器实际使用的网络策略之一，实现手段是：使用网桥docker0并创建veth pair虚拟网络接口对，最终使一个veth附加在宿主机的docker0网桥之上，而另一个veth安置在容器的net namespace内部。

libcontainer中实现veth接口的代码非常通俗易懂，位于./docker/libcontainer/network/veth.go#L19-L50，如下所示：

```
name1, name2, err := createVethPair(prefix)
if err != nil {
     return err
}
if err := SetInterfaceMaster(name1, bridge); err != nil {
     return err
}
if err := SetMtu(name1, n.Mtu); err != nil {
     return err
}
if err := InterfaceUp(name1); err != nil {
     return err
}
if err := SetInterfaceInNamespacePid(name2, nspid); err != nil {
     return err
}
```

主要的流程包含以下四个步骤：

1) 在宿主机上创建veth pair。

2) 将一个veth附加至docker0网桥上。

3) 启动第一个veth。

4） 将第二个veth附加至libcontainer创建进程的namespace下（ 注意：并未启动第二个veth） 。

使用Create函数实现veth pair的创建之后，libcontainer在Initialize函数中实现将网络namespace中的veth改名为“eth0”、设置网络设备的MTU、以及启动网络接口等。

3.netns网络栈的创建

netns针对Docker容器的other container网络模式服务。netns完成的工作是：将其他容器的net namespace路径，传递给需要创建other container网络模式的容器使用。

libcontainer中实现netns策略的源码位于./docker/libcontainer/network/netns.go#L17-L20，如下所示：

```
func (v *NetNS) Create(n *Network, nspid int, networkState *NetworkState) error {
     networkState.NsPath = n.NsPath
     return nil
}
```

libcontainer使用Create函数先将NsPath传递给新建容器，再在Initialize函数中实现将net namespace的文件描述符交由新创建容器使用，最终实现两个Docker容器共享同一个网络栈。

通过Create函数，Docker容器相应的网络栈环境即已经完成创建，初始化工作有待Initialize函数来完成。详见本书第13章，有关dockerinit的执行将完成容器网络栈的初始化。

7.7 总结

如何使用Docker容器的网络，一直是工业界关心的问题。本章从Linux内核原理的角度阐述了什么是Docker容器，并对Docker容器的4种网络模式进行了初步的介绍，最终贯穿Docker架构中的多个模块，如Docker Client、Docker Daemon、execdriver以及libcontainer，深入分析了Docker容器网络的实现。

目前，若只谈论Docker，那么它还是只停留在单宿主机的场景上。如何面对跨宿主机的场景、如何实现分布式Docker容器的管理，目前为止还没有一个一劳永逸的解决方案。再者，一个解决方案的存在，总是会适应于一个应用场景。Docker这种容器技术的发展，大大改善了传统模式下使用诸如虚拟机等传统计算单位存在的多种弊端，却在网络方面使得自身的使用存在瑕疵。希望本章是一个引子，介绍Docker容器网络，从源码的角度分析Docker容器网络之后，能使更多的Docker爱好者思考Docker容器网络的来龙去脉，并为Docker乃至容器技术的发展做出贡献。

第8章　Docker镜像

8.1　引言

2014年，Docker便在全球刮起了一阵又一阵的“容器风”，全球的开发者开始认识Docker，学习Docker。又过了一年，工业界对Docker的态度已经不再是了解与观望，转而是实践一波高过一波。Docker目 前的发展似乎并不会像其他昙花一现的技术一样，反而是凭借创新性的特性，Docker在工业界经过实践与评估之后，显现了前所未有的潜力。

究其本质，“Docker提供容器服务”这句话，相信很少有人会有异议。既然如此，Docker提供的服务属于“容器”技术，那么反观“容器”技术的本质与发展历史，我们又可以发现什么呢？正如第7章所提到的，Docker使用的“容器”技术，主要是以Linux内核的namespace、cgroup等特性为基础，保障进程或者进程组处于一个隔离、受限、安全的环境之中。Docker第一个版本在2013年3月发行，而cgroups正式在Linux操作系统中的亮相可以追溯到2007年下半年，当时cgroups被合并至Linux内核2.6.24版本。这整整6年时间并不是Linux平台“容器”技术发展的真空期。2008年LXC(　Linux Container)　诞生，它简化了容器的创建与管理；之后业界一些PaaS

平台也初步尝试采用容器技术作为其云应用的运行环境。而在Docker发布的同年，Google也发布了开源容器管理工具lmctfy。除此之外，若抛开Linux操作系统，其他操作系统（如FreeBSD、Solaris等）同样诞生了作用类似的“容器”技术，其发展历史更是可以追溯至千禧年初期。

总之，“容器”技术的发展不可谓短暂，然而论同时代的影响力，却鲜有Docker的媲美者。不论是云计算大潮催生了Docker技术，抑或是Docker技术赶上了云计算的大时代，毋庸置疑的是，Docker作为技术领域的新宠儿，必将继续受到业界的广泛青睐。云计算时代，分布式应用逐渐流行，大部分应用对自身的构建、交付与运行都有着与传统不一样的需求。借助Linux内核的namespace和cgroup等特性，自然可以实现应用运行环境的资源隔离与限制等；然而，namespace和cgroup等内核特性却无法为容器的运行环境做全盘打包。而Docker的设计则非常巧妙地考虑到了这一点，除namespace和cgroup之外，Docker另外采用了神奇的“镜像”技术作为Docker管理文件系统以及运行环境强有力的补充。Docker灵活的“镜像”技术，在笔者看来，也是其大红大紫最重要的因素之一。

8.2　Docker镜像介绍

Docker诞生至今，很多容器领域从业者都会将Docker技术与虚拟机技术相提并论。提及Docker镜像，大家肯定也会联想到虚拟机中的镜像。镜像是一种文件存储形式，文件管理员可以通过技术手段将很多文件制作成一个镜像。对于虚拟机而言，镜像文件中存储着操作系统、文件系统内容、设备文件等。可以说，Docker镜像与虚拟机镜像有很大的相似度，然而也有着本质的区别。相似的是，两者存储内容大致相同，都会含有文件系统内容；不同的是，Docker镜像不含操作系统内容，同时Docker镜像由多个镜像组成。

根据Docker官方网站上的技术文档描述，image（镜像）是Docker术语的一种，对于容器而言，它代表一个只读的layer。而layer则具体代表Docker容器文件系统中可叠加的一部分。

如此介绍Docker镜像，相信众多Docker爱好者理解起来仍然是云里雾里。那么理解之前，先让我们来认识一下与Docker镜像相关的4个概念：rootfs、union mount、image以及layer。

8.3　rootfs

rootfs代表一个Docker容器在启动时（而非运行后）其内部进程可见的文件系统视角，或者Docker容器的根目录。当然，该目录下含有Docker容器所需要的系统文件、工具、容器文件等。

传统上，Linux操作系统内核启动时，内核首先会挂载一个只读（read-only）的rootfs，当系统检测其完整性之后，决定是否将其切换为读写（read-write）模式，或者最后在rootfs之上另行挂载一种文件系统并忽略rootfs。Docker架构下，依然沿用Linux中rootfs的思想。当Docker Daemon为Docker容器挂载rootfs的时候，与传统Linux内核类似，将其设定为只读模式。在rootfs挂载完毕之后，和Linux内核不一样的是，Docker Daemon没有将Docker容器的文件系统设为读写模式，而是利用Union Mount的技术，在这个只读的rootfs之上再挂载一个读写的文件系统，挂载时该读写文件系统内空无一物。在这里，我们暂且把Docker容器的文件系统这么理解：只含只读的rootfs和可读写的文件系统。

举一个Ubuntu容器启动的例子。假设用户已经通过Docker Registry下拉了Ubuntu：14.04的镜像，并通过命令docker run–it ubuntu：14.04/bin/bash将其启动并运行，则Docker Daemon为其创建的rootfs以及容器可读写的文件系统如图8-1所示。

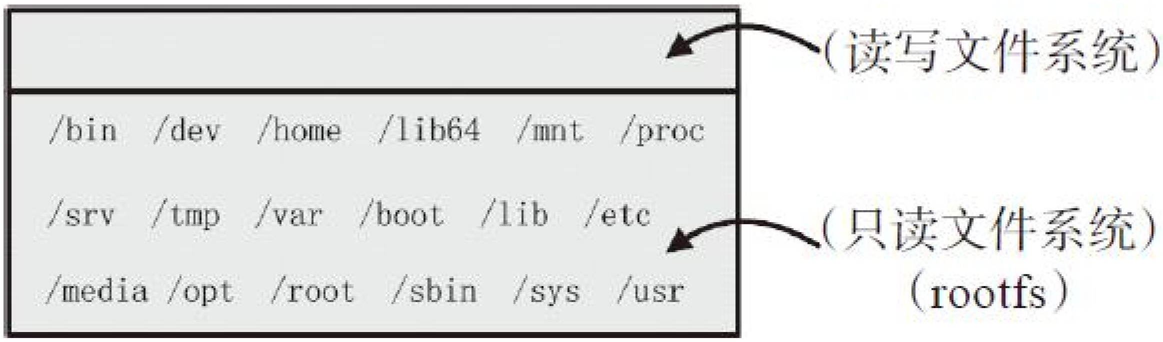


图8-1　Ubuntu 14.04容器文件系统示意图

顾名思义，该容器中的进程对rootfs中的内容只拥有读权限，对于读写文件系统中的内容既拥有读权限也拥有写权限。通过观察图8-1可以发现：容器虽然只有一个文件系统，但该文件系统由“两层”组成，分别为读写文件系统和只读文件系统。通过这样的理解，文件系统已然有了层级（layer）的意味。

简单来讲，可以将Docker容器的文件系统分为两部分，而前面提到的Docker Daemon利用Union Mount技术，将两者挂载。那么Union Mount又是一种怎样的技术？下一节将介绍Union Mount的概念。

8.4 Union Mount

Union Mount代表一种文件系统挂载方式，允许同一时刻多种文件系统叠加挂载在一起，并以一种文件系统的形式，呈现多种文件系统内容合并后的目录。

一般情况下，若通过某种文件系统挂载内容至挂载点，挂载点目录中原先的内容将会被隐藏。而Union Mount则不会将挂载点目录中的内容隐藏，反而是将挂载点目录中的内容和被挂载的内容合并，并为合并后的内容提供一个统一独立的文件系统视角。通常来讲，被合并的文件系统中只有一个会以读写（read-write）模式挂载，其他文件系统的挂载模式均为只读（read-only）。实现这种Union Mount技术的文件系统一般称为联合文件系统（Union Filesystem），较为常见的有UnionFS、aufs、OverlayFS等。

Docker实现容器文件系统Union Mount时，提供多种具体的文件系统解决方案，如Docker早期版本沿用至今的AUFS，还有在Docker 1.4.0版本中开始支持的OverlayFS等。

为了更深入地了解Union Mount，可以使用aufs文件系统来进一步阐述本章前面Ubuntu：14.04容器文件系统的例子，挂载Ubuntu 14.04文件系统的示意图如图8-2所示。

使用镜像Ubuntu：14.04创建的容器中，可以暂且将该容器的整个rootfs当成一个文件系统。前面也提到，挂载时读写文件系统中空无一物。既然如此，从用户视角来看，容器内文件系统和rootfs完全一样，用户完全可以按照往常习惯，无差别地使用自身视角下文件系统中的所有内容；然而，从内核的角度来看，两者有非常大的区别。追溯区别存在的根本原因，那就不得不提及aufs等文件系统的COW（copy-on-write）特性。

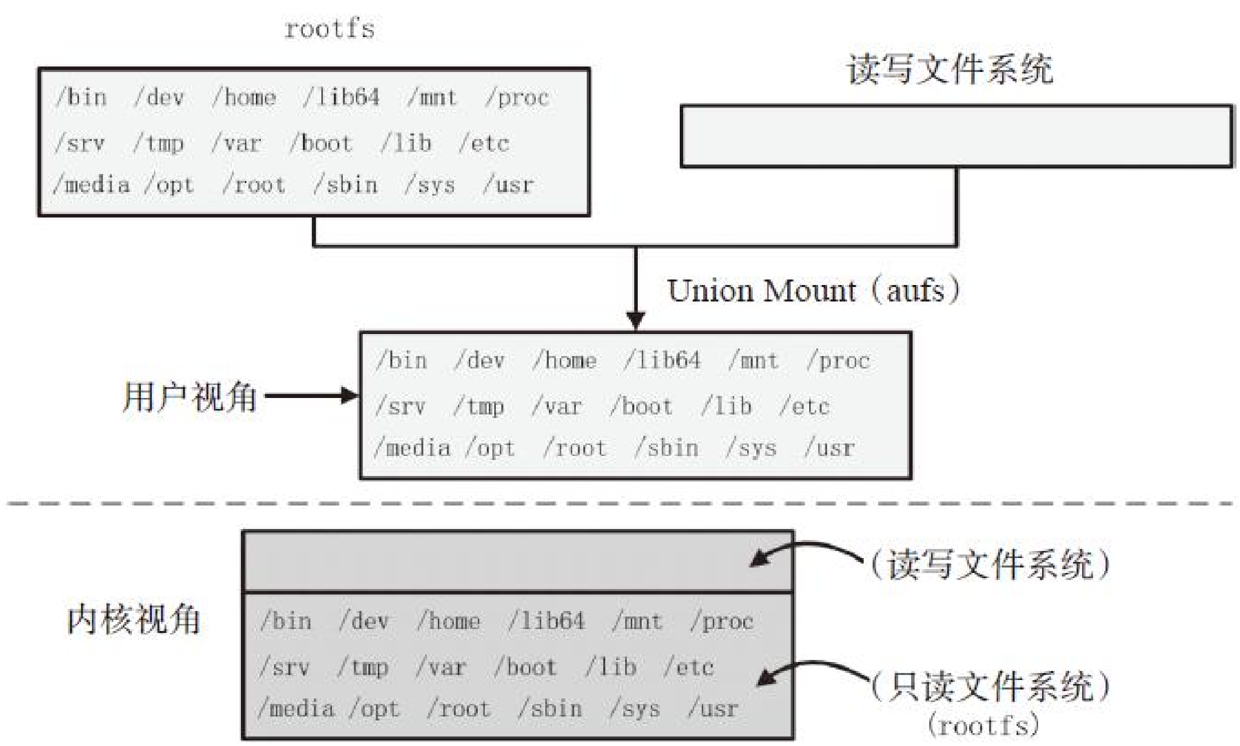


图8-2　aufs挂载Ubuntu 14.04文件系统的示意图

COW文件系统和其他文件系统最大的区别就是：前者从不覆写已有文件系统中已有的内容。通过COW文件系统将两个文件系统（rootfs和读写文件系统）合并，最终用户视角为合并后含有所有内容

的文件系统，然而在Linux内核逻辑上依然可以区别两者，那就是用户对原先rootfs中的内容拥有只读权限，而对读写文件系统中的内容拥有读写权限。

既然对用户而言，全然不知哪些内容只读，哪些内容可读写，这些信息只有内核在接管，那么假设用户需要更新其视角下的文件/etc/bash.bashrc，而该文件又恰巧是rootfs只读文件系统中的内容，内核是否会抛出异常或者驳回用户请求呢？答案是否定的。当此情形发生时，COW文件系统首先不会覆写只读文件系统中的文件，即不会覆写rootfs中的/etc/bash.bashrc，其次反而会将该文件复制至读写文件系统中，即将/etc/bash.bashrc复制至读写文件系统中的/etc/bash.bashrc（此时，rootfs文件系统和读写文件系统中各含有一份/etc/bash.bashrc），最后再对后者进行更新操作。如此一来，纵使rootfs与读写文件系统中均有/etc/bash.bashrc，诸如aufs类型的COW文件系统也能保证用户视角中只能看到读写文件系统中的/etc/bash.bashrc，即更新后的内容。

当然，这样的特性同样支持rootfs中文件的删除等其他操作。例如：用户通过apt-get软件包管理工具安装Golang，所有与Golang相关的内容都会安装在读写文件系统中，而不会安装在rootfs中。此时用户又希望通过apt-get软件包管理工具删除所有关于MySQL的内容，恰巧这部分内容又都存在于rootfs中，删除操作执行时同样不会删除

rootfs实际存在的MySQL，而是在读写文件系统中删除该部分内容，导致最终rootfs中的MySQL对容器用户不可见，也不可访。由于读写文件系统中根本不存在MySQL的相关内容，因此似乎在读写文件系统中找不到需要删除的对象。此时，aufs保障在读写文件系统中对这些文件内容做相关的标记（whiteout），确保用户在查看文件系统内容时，读写文件系统中的whiteout将遮盖住rootfs中的相应内容，导致这些内容不可见，以达到与删除这部分内容相类似的效果。

掌握Docker中rootfs以及Union Mount的概念之后，再来理解Docker镜像，就会有水到渠成的感觉。

8.5 image

Docker中rootfs的概念，起到容器文件系统中基石的作用。对于容器而言，其只读的特性也是不难理解。神奇的是，实际情况下Docker架构中rootfs的设计与实现比前面的描述还要精妙得多。

继续以Ubuntu 14.04为例，虽然通过aufs可以实现rootfs与读写文件系统的合并，但是考虑到rootfs自身接近200MB的磁盘大小，如果以这个rootfs的粒度来实现容器的创建与迁移等，是否会稍显笨重？同时也会大大降低镜像的灵活性？而且，若用户希望拥有一个Ubuntu 14.10的rootfs，那么是否有必要创建一个全新的rootfs，毕竟Ubuntu 14.10和Ubuntu 14.04的rootfs中有很多一致的内容。

Docker中image的概念，非常巧妙地解决了以上问题。最为简单地解释image，它就是Docker容器中只读文件系统rootfs的一部分。换言之，实际上Docker容器的rootfs可以由多个image来构成。多个image构成rootfs的方式依然沿用Union Mount技术。

多个image构成的rootfs如图8-3所示（其中，rootfs中每一层image中的内容划分只为了阐述清楚rootfs由多个image构成，并不代表实际情况下rootfs中的内容划分）：

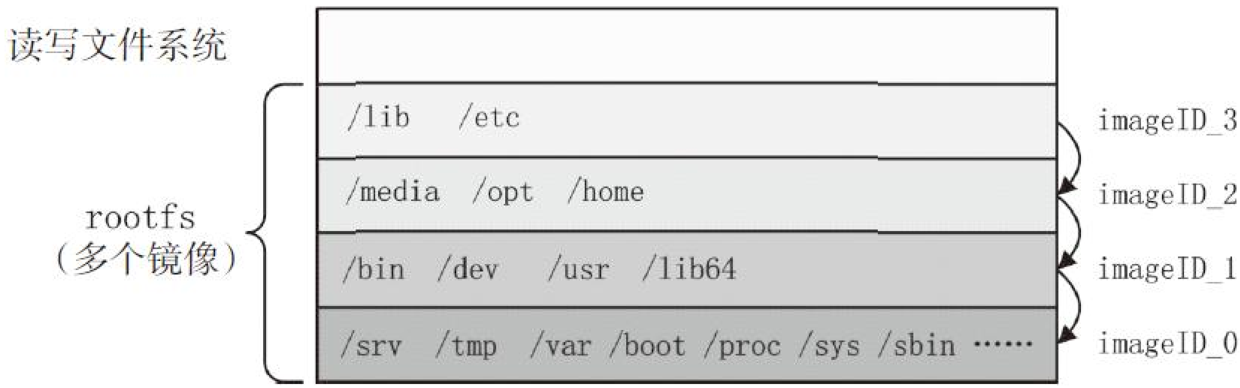


图8-3　容器rootfs多image构成图

从图8-3中我们可以看出，容器rootfs包含4个image，其中每个image中都有一些用户视角文件系统中的一部分内容。4个image处于层叠的关系，除了最底层的image，每一层的image都叠加在另一个image之上。另外，每一个image均含有一个image ID，用于唯一地标记该image。

基于以上概念，Docker Image中又抽象出两种概念：父镜像以及基础镜像。除了容器rootfs最底层的镜像，其余镜像都依赖于其底下的一个或多个镜像。这种情况下，Docker将下一层的镜像称为上一层镜像的父镜像。以图9-3为例，imageID_0是imageID_1的父镜像，imageID_2是imageID_3的父镜像，而imageID_0没有父镜像。对于最下层的镜像，即没有父镜像的镜像，在Docker中我们习惯称之为基础镜像。

通过image的形式，原先较为臃肿的rootfs被逐渐打散成轻便的多层。除了轻便的特性之外，image同时还有前面提到的只读特性，

如此一来，在不同的容器、不同的rootfs中image完全可以用来复用。

多image组织关系与复用关系如图8-4所示（图中镜像名称的举例只为将image之间的关系阐述清楚，并不代表实际情况下相应名称image之间的关系）：

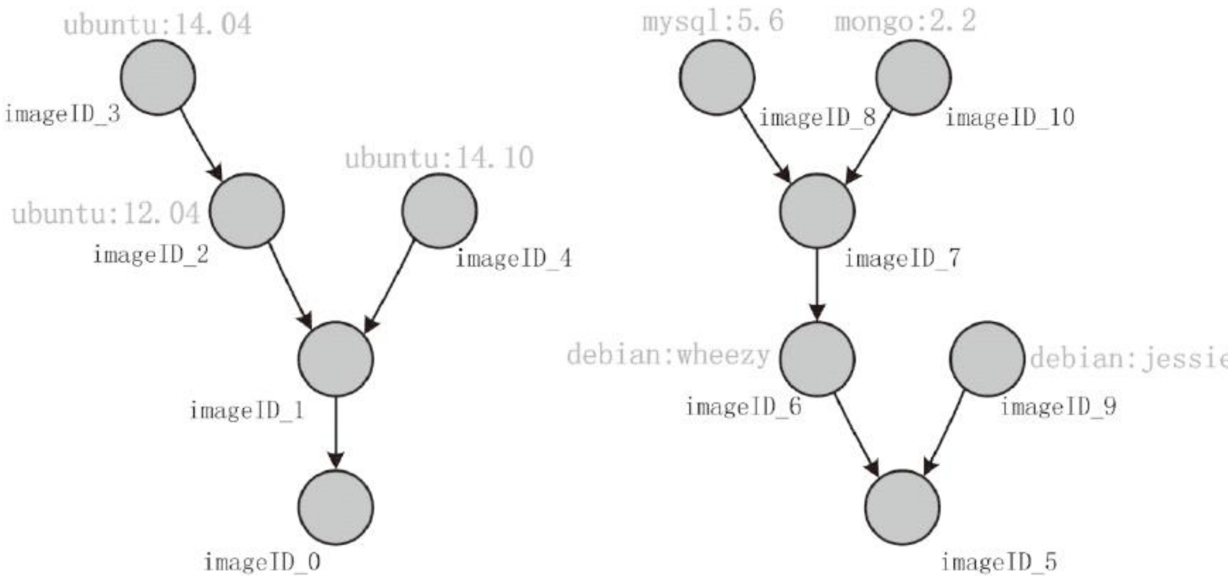


图8-4　多image组织关系示意图

图8-4中，共罗列了11个image，这11个image之间的关系呈现为一幅森林图。森林中含有两棵树，左边树中包含5个节点，即含有5个image；右边树中包含6个节点，即含有6个image。其中，有些image标记了加粗的字段，这意味该image代表某一种容器镜像rootfs的最上层image。如ubuntu：14.04，代表imageID_3为该类型容器rootfs的最上层，沿着该节点找到树的根节点，可以发现路径上

还有imageID_2、imageID_1和imageID_0。特殊的是，imageID_2作为imageID_3的父镜像，同时又是容器镜像ubuntu：12.04的rootfs中的最上层，可见镜像ubuntu：14.04只是在镜像ubuntu：12.04之上再另行叠加了一层。因此，在下载镜像ubuntu：12.04以及ubuntu：14.04时，只会下载一份imageID_2、imageID_1和imageID_0，实现image的复用。同时，右边树中mysql：5.6、mongo：2.2、debian：wheezy和debian：jessie也呈现同样的关系。

8.6 layer

在Docker中，术语layer是一个与image含义较为相近的词。容器镜像的rootfs是容器只读的文件系统，rootfs又由多个只读的image构成。于是，rootfs中每个只读的image都可以称为一个layer。

除了只读的image之外，Docker Daemon在创建容器时会在容器的rootfs之上，再挂载一层读写文件系统，而这一层文件系统也称为容器的一个layer，常被称为top layer。实际情况下，Docker还会在rootfs和top layer之间再挂载一个layer，这一个layer中主要包含的内容是/etc/hosts、/etc/hostname以及/etc/resolv.conf，一般这一个layer称为init layer。为了简化阐述流程，我们暂不提init layer。

因此，总之，Docker容器中每一层只读的image以及最上层可读写的文件系统，均称为layer。如此一来，layer的范畴比image多了一层，即多包含了最上层的读写文件系统。

有了layer的概念，大家可以思考这样一个问题：容器文件系统分为只读的rootfs，以及可读写的top layer，那么容器运行时若在top

layer中写入了内容，这些内容是否可以持久化，并且也被其他容器复用？

本章对于image的分析中，提到了image有复用的特性，既然如此，再提出一个更为大胆的假设：容器的top layer是否可以转变为image？

答案是肯定的。Docker的设计理念中，top layer转变为image的行为（Docker中称为commit操作）进一步释放了容器rootfs的灵活性。Docker的开发者完全可以基于某个镜像创建容器做开发工作，并且无论在开发周期的哪个时间点，都可以对容器进行commit，将所有top layer中的内容打包为一个image，构成一个新的镜像。commit完毕之后，用户完全可以基于新的镜像，进行开发、分发、测试、部署等。不仅Docker commit的原理如此，基于Dockerfile的docker build最为核心的思想，也是不断将容器的top layer转化为image。

8.7 总结

Docker风暴席卷全球并非偶然。如今的云计算时代下，轻量级容器技术与灵活的镜像技术相结合，似乎颠覆了以往的软件交付模式，为持续集成（Continuous Integration，CI）与持续交付（Continuous Delivery，CD）的发展带来了全新的契机。

理解Docker的“镜像”技术，有助于Docker爱好者更好地使用、创建以及交付Docker镜像。基于此，本章从Docker镜像的4个重要概念入手，介绍了Docker镜像中包含的内容，涉及的技术，以及重要的特性。Docker引入优秀的“镜像”技术时，着实使容器的使用变得更为便利，也拓宽了Docker的使用范畴。然而，与此同时，我们也应该理性地看待镜像技术引入时，是否会带来其他副作用，如镜像技术是否会带来性能问题等都将是我们进一步思考的话题。

第9章　Docker镜像下载

9.1　引言

说Docker Image是Docker体系的价值所在，没有丝毫的夸张。Docker Image作为容器运行环境的基石，彻底解放了Docker容器的生命力，也激发了用户对于容器运用的无限想象力。

玩转Docker，必然离不开Docker Image的支持。然而“万物皆有源”，Docker Image来自何方，Docker Image又是通过何种途径传输到用户的宿主机，以致用户可以通过Docker Image创建容器的呢？回忆初次接触Docker的场景，大家肯定对两条命令不陌生：docker pull和docker run。这两条命令中，正是前者实现了Docker Image的下载。Docker Daemon在执行这条命令时，会将Docker Image从Docker Registry下载至本地，并保存在本地Docker Daemon管理的Graph中。

谈及Docker Registry，Docker爱好者首先联想到的自然是Docker Hub。Docker Hub作为Docker官方支持的Docker Registry，拥有全球成千上万的Docker Image。全球的Docker爱好者除了可以下载Docker Hub开放的镜像资源之外，还可以向

Docker Hub贡献镜像资源。在Docker Hub上，用户不仅可以享受公有镜像带来的便利，而且可以创建私有镜像库。Docker Hub是全国最大的Public Registry，另外Docker还支持用户自定义创建Private Registry。Private Registry主要的功能是为私有网络提供Docker镜像的专属服务，一般而言，镜像种类适应用户需求，私密性较高，且不会占用公有网络带宽。

本章主要从Docker 1.2.0源码的角度分析Docker下载Docker Image的过程。分析内容主要包括以下4部分：

1）概述Docker镜像下载的流程，涉及Docker Client、Docker Server以及Docker Daemon；

2）Docker Client处理并发送docker pull请求；

3）Docker Server接收docker pull请求，创建镜像下载任务并触发执行；

4）Docker Daemon执行镜像下载任务，并存储镜像至Graph。

9.2　Docker镜像下载流程

Docker Image作为Docker生态中的精髓，下载过程中需要Docker架构中多个组件的协作，下载流程如图9-1所示。

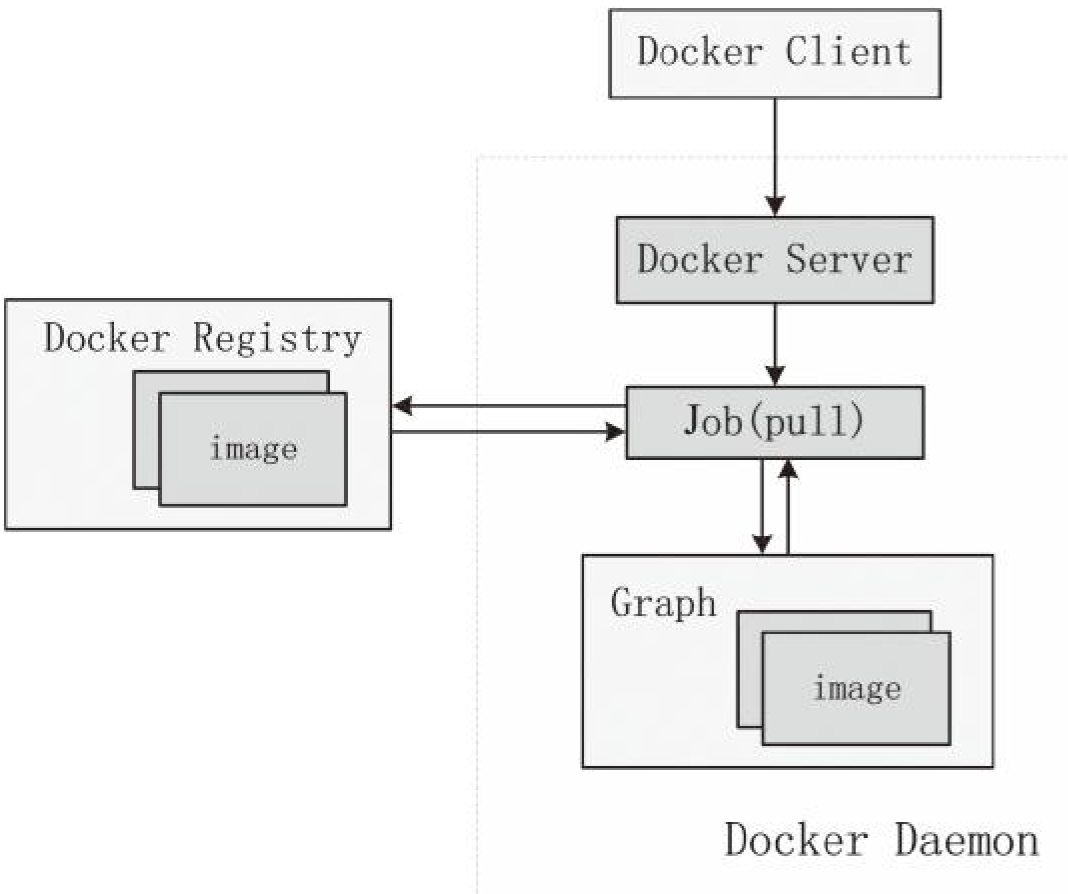


图9-1　Docker镜像下载流程图

如图9-1所示，Docker Image的下载流程可以归纳为以下3个步骤：

1）用户通过Docker Client发送pull请求，用于让Docker Daemon下载指定名称的镜像；

2）Docker Server接收Docker镜像的pull请求，创建下载镜像任务并触发执行；

3）Docker Daemon执行镜像下载任务，从Docker Registry中下载指定镜像，并将其存储于本地的Graph中。

9.3 Docker Client

Docker架构中，Docker用户的角色绝大多数由Docker Client来扮演。因此，用户对Docker的管理请求全部由Docker Client来发送，Docker镜像下载请求自然也不例外。

为了更清晰地描述Docker镜像下载，本节结合具体的命令进行分析，命令如下：

```
docker pull ubuntu:14.04
```

此命令的含义是：通过docker二进制可执行文件，执行镜像下载的pull命令，镜像参数为ubuntu：14.04，镜像名称为ubuntu，镜像标签（tag）为14.04。此命令一经发起，第一个接受并处理的Docker组件为Docker Client，执行内容包括以下三个步骤：

1）解析命令中与Docker镜像相关的参数；

2）配置Docker下载镜像时所需的认证信息；

3）发送RESTful请求至Docker Daemon。

9.3.1　解析镜像参数

通过docker二进制文件执行docker pull ubuntu：14.04时，Docker Client首先会创建，随后通过参数处理分析出请求类型pull，最终执行pull请求相应的处理函数。关于Docker Client的创建与命令执行可以参见第2章。

Docker Client执行pull请求相应的处理函数，源码位于./docker/api/client/command.go#L1183-L1244，有关提取镜像参数的源码如下：

```
func (cli *DockerCli) CmdPull(args ...string) error {
		cmd := cli.Subcmd("pull", "NAME[:TAG]", "Pull an image or a repository 
from the registry")
		tag := cmd.String([]string{"#t", "#-tag"}, "", "Download tagged image 
in a repository")
	if err := cmd.Parse(args); err != nil {
		return nil
	}
	if cmd.NArg() != 1 {
		cmd.Usage()
		return nil
	}
	var (
		v      = url.Values{}
		remote = cmd.Arg(0)
	)
	v.Set("fromImage", remote)
	if *tag == "" {
		v.Set("tag", *tag)
	}
	remote, _ = parsers.ParseRepositoryTag(remote)
	// Resolve the Repository name from fqn to hostname + name
	hostname, _, err := registry.ResolveRepositoryName(remote)
	if err != nil {
		return err
	}
	...
}
```

结合命令docker pull ubuntu：14.04来分析CmdPull函数的定义，我们可以发现，该函数传入的形参为args，实参是字符串ubuntu：14.04。另外，纵观以上源码，可以发现Docker Client解析的镜像参数无外乎4个：tag、remote、v以及hostname，四者各自的作用如下。

·tag：带有Docker镜像的标签（已弃用）；

·remote：带有Docker镜像的名称与标签；

·v：类型为url.Values，实质是一个map类型，用于配置请求中URL的查询参数；

·hostname：Docker Registry的地址，代表用户希望从指定的Docker Registry下载Docker镜像。

1.解析tag参数

Docker镜像的tag参数，是第一个被Docker Client解析的镜像参数，代表用户所需下载Docker镜像的标签信息，如docker pull ubuntu：14.04请求中镜像的tag信息为14.04，若用户使用docker pull ubuntu请求下载镜像，没有显式指定tag信息时，Docker Client会默认该镜像的tag信息为latest。

Docker 1.2.0版本除了以上的tag信息传入方式之外，依旧保留着代表镜像标签的flag参数tag，而这个flag参数在1.2.0版本的使用过程中已经弃用，并会在之后新版本的Docker中被移除，因此在使用Docker 1.2.0版本下载Docker镜像时，不建议使用flag参数tag。传入tag信息的方式，建议使用docker pull NAME[：TAG]的形式。

关于Docker 1.2.0版本依旧保留的flag参数tag，其定义与解析的源码位于：./docker/api/client/commands.go#1185-L1188，如下：

```
tag := cmd.String([]string{"#t", "#-tag"}, "", "Download tagged image in a
repository")
if err := cmd.Parse(args); err != nil {
     return nil
}
```

以上源码说明：CmdPull函数解析tag参数时，Docker Client首先定义一个flag参数，flag参数的名称为“#t”或者“#-tag”，用途为：指定Docker镜像的tag参数，默认值为空字符串；随后通过cmd.Parse(args) 的执行，解析args中的tag参数。

2.解析remote参数

Docker Client解析完tag参数之后，同样需要解析出Docker镜像所属的repository，如请求docker pull ubuntu：14.04中，

Docker镜像为ubuntu：14.04，镜像的repository信息为ubuntu，镜像的tag信息为14.04。

Docker Client通过解析remote参数，使得remote参数携带repository信息和tag信息。Docker Client解析remote参数的第一个步骤，源码如下：

```
remote = cmd.Arg(0)
```

其中，cmd的第一个参数赋值给remote，以docker pull ubuntu：14.04为例，cmd.Arg（0）为ubuntu：14.04，则赋值后remote值为ubuntu：14.04。此时remote参数不仅包含Docker镜像的repository信息，还包含tag信息。若用户请求中带有Docker Registry的信息，如docker pull localhost.localdomain：5000/docker/ubuntu：14.04，cmd.Arg（0）为localhost.localdomain：5000/docker/ubuntu：14.04，则赋值后remote值为localhost.localdomain：5000/docker/ubuntu：14.04，此时remote参数同时包含Docker Registry信息、repository信息以及tag信息。

随后，在解析remote参数的第二个步骤中，Docker Client通过解析赋值完毕的remote参数解析出repository信息，并再次覆写remote参数的值，源码如下：

```
remote, _ = parsers.ParseRepositoryTag(remote)
```

ParseRepositoryTag的作用是：解析出remote参数的repository信息和tag信息，该函数的实现位于./docker/pkg/parsers/parsers.go#L72-L81，源码如下：

```
func ParseRepositoryTag(repos string) (string, string) {
     n := strings.LastIndex(repos, ":")
     if n < 0 {
          return repos, ""
     }
     if tag := repos[n+1:]; !strings.Contains(tag, "/") {
          return repos[:n], tag
     }
     return repos, ""
}
```

以上函数的实现过程充分考虑了多种不同Docker Registry的情况，如：请求docker pull ubuntu：14.04中remote参数为ubuntu：14.04，而请求docker pull localhost.localdomain：5000/docker/ubuntu：14.04中用户指定了Docker Registry的地址localhost.localdomain：5000/docker，故remote参数还携带了Docker Registry信息。

ParseRepositoryTag函数首先从repos参数的尾部往前寻找“：”，若不存在，则说明用户没有显式指定Docker镜像的tag，返回整个repos作为Docker镜像的repository；若“：”存在，则说明用户显式指定了Docker镜像的tag，“：”前的内容作为repository信息，“：”后的内容作为tag信息，并返回两者。

ParseRepositoryTag函数执行完，回到CmdPull函数，返回内容的repository信息将覆写remote参数。对于请求docker pull localhost.localdomain：5000/docker/ubuntu：14.04，remote参数被覆写后，值为localhost.localdomain：5000/docker/ubuntu，携带Docker Registry信息以及repository信息。

3.配置url.Values

Docker Client发送请求给Docker Server时，需要为请求配置URL的查询参数。CmdPull函数的执行过程中创建url.Value并配置的源码实现位于./docker/api/client/commands.go#L1194-L1203，如下：

```
var (
     v      = url.Values{}
     remote = cmd.Arg(0)
)
v.Set("fromImage", remote)
if *tag == "" {
     v.Set("tag", *tag)
}
```

其中，变量v的类型是url.Values，最终为URL配置的查询参数有两个，分别为“fromImage”与“tag”，“fromImage”的值是remote参数没有被覆写时的值，“tag”的值一般为空，原因是tag参数已弃用，一般不使用flag参数tag。

4.解析hostname参数

Docker Client解析镜像参数时，还有一个重要的环节，那就是解析Docker Registry的地址信息。若用户希望从指定的Docker Registry中下载Docker镜像，则Docker Client需要考虑这种情况，为用户解析出Docker Registry的地址。

解析Docker Registry地址的代码实现位于./docker/api/client/commands.go#L1207，如下：

```
hostname, _, err := registry.ResolveRepositoryName(remote)
```

Docker Client通过包registry中的函数ResolveRepositoryName来解析hostname参数，传入的实参为remote，即去tag化的remote参数。ResolveRepositoryName函数的源码实现位于./docker/registry/registry.go#L237-L259，如下：

```
func ResolveRepositoryName(reposName string) (string, string, error) {
     if strings.Contains(reposName, "://") {
          // It cannot contain a scheme!
          return "", "", ErrInvalidRepositoryName
     }
     nameParts := strings.SplitN(reposName, "/", 2)
     if len(nameParts) == 1
     || (!strings.Contains(nameParts[0], ".") && !strings.Contains(nameParts[0],
":") &&
     nameParts[0] != "localhost") {
          // This is a Docker Index repos (ex: samalba/hipache or ubuntu)
          err := validateRepositoryName(reposName)
          return IndexServerAddress(), reposName, err
     }
     hostname := nameParts[0]
```

```
        reposName = nameParts[1]
        if strings.Contains(hostname, "index.docker.io") {
                return "", "", fmt.Errorf("Invalid repository name, try \"%s\"
instead", reposName)
        }
        if err := validateRepositoryName(reposName); err != nil {
            return "", "", err
        }
            return hostname, reposName, nil
}
```

ResolveRepositoryName函数首先通过“/”分割字符串reposName，如下：

```
nameParts := strings.SplitN(reposName, "/", 2)
```

如果nameParts的长度为1，则说明reposName中不含有字符“/”，这意味着用户没有指定Docker Registry。另外，形如“samalba/hipache”的reposName同样说明用户并没有指定Docker Registry，其中samalba为用户在Docker Hub上的用户名。当用户没有指定Docker Registry时，Docker Client默认返回IndexServerAddress（），该函数返回常量INDEXSERVER，值为“https://index.docker.io/v1”。也就是说，当用户下载Docker镜像时，若不指定Docker Registry，默认情况下，Docker Client通知Docker Daemon从Docker Hub上下载镜像。例如：请求docker pull ubuntu：14.04，由于没有指定Docker Registry，Docker Client默认使用全球最大的Docker Registry——Docker Hub。

当不满足返回默认Docker Registry时，Docker Client通过解析reposNames，得出用户指定的Docker Registry地址。例如：请求docker pull localhost.localdomain：5000/docker/ubuntu：14.04中，解析出的Docker Registry地址为localhost.localdomain：5000。

至此，与Docker镜像相关的参数已经全部解析完毕，Docker Client将携带这部分重要信息以及用户的认证信息来构建RESTful请求，发送给Docker Server。

9.3.2　配置认证信息

用户下载Docker镜像时，Docker同样支持用户信息的认证。用户认证信息由Docker Client配置；Docker Client发送请求至Docker Server时，用户认证信息也被一并发送；随后，Docker Daemon处理下载Docker镜像的请求时，用户认证信息将在Docker Registry中验证。

Docker Client配置用户认证信息包含两个步骤，实现源码如下：

```
cli.LoadConfigFile()
// Resolve the Auth config relevant for this server
authConfig := cli.configFile.ResolveAuthConfig(hostname)
```

可见，第一个步骤是使cli（Docker Client）加载ConfigFile，ConfigFile是Docker Client用于存放用户在Docker Registry上认证信息的对象。DockerCli、ConfigFile以及AuthConfig三种数据结构之间的关系如图9-2所示。

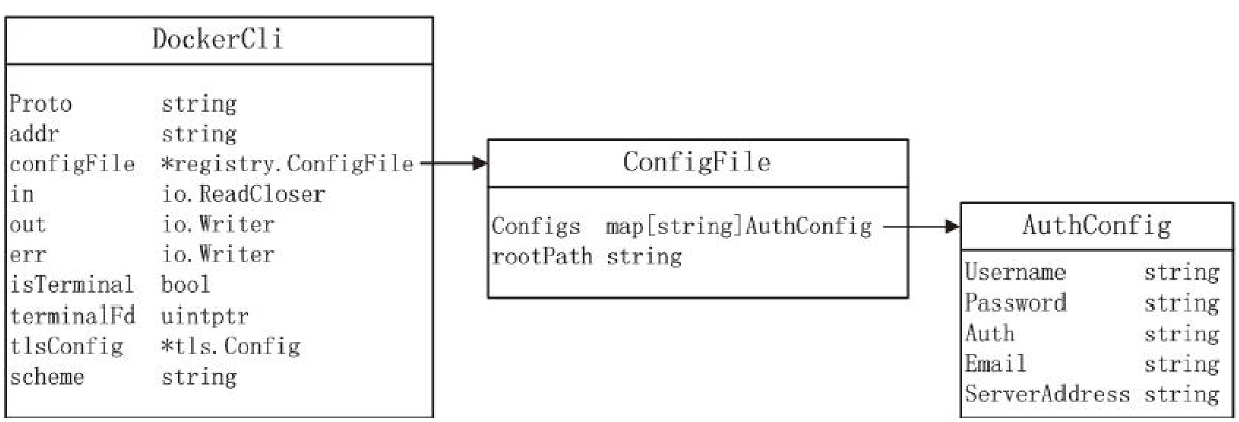


图9-2　DockerCli、ConfigFile以及AuthConfig关系图

DockerCli结构体的属性configFile是一个指向registry.ConfigFile的指针，而ConfigFile结构体的属性Configs属于map类型，其中key为string，代表Docker Registry的地址，value的类型为AuthConfig。AuthConfig类型的具体含义为用户在某个Docker Registry上的认证信息，包含用户名、密码、认证信息、邮箱地址以及服务器地址等。

加载完用户所有的认证信息之后，Docker Client配置用户认证信息的第二个步骤是：通过用户指定的Docker Registry，即之前解析出的hostname参数，从用户所有的认证信息中找出与指定hostname相匹配的认证信息。新创建的authConfig，类型即为AuthConfig，将会作为用户在指定Docker Registry上的认证信息，发送至Docker Server。

9.3.3　发送API请求

解析完所有的Docker镜像参数，并且配置完毕用户的认证信息之后，Docker Client需要使用这些信息正式发送镜像下载的请求至Docker Server。

Docker Client定义了pull函数来实现发送镜像下载请求至Docker Server，源码位于./docker/api/client/commands.go#L1217-L1229，如下：

```
pull := func(authConfig registry.AuthConfig) error {
	buf, err := json.Marshal(authConfig)
	if err != nil {
		return err
	}
	registryAuthHeader := []string{
		base64.URLEncoding.EncodeToString(buf),
	}
		return cli.stream("POST", "/images/create?"+v.Encode(), nil, cli.out, 
map[string][]string{
		"X-Registry-Auth": registryAuthHeader,
	})
}
```

pull函数的实现较为简单，首先通过authConfig对象创建registryAuthHeader，最后发送POST请求，请求的URL为“/images/create？”+v.Encode（），在URL中传入查询参数，包括“fromImage”与“tag”，另外在请求的HTTP Header中添加认证信息registryAuthHeader。

执行以上pull函数时，Docker镜像下载请求被发送，随后Docker Client等待Docker Server的接收、处理与响应。

9.4 Docker Server

Docker Server作为Docker Daemon的入口，所有Docker Client发送请求都由Docker Server接收。Docker Server通过解析请求的URL与请求方法，最终路由分发至相应的处理方法来处理。Docker Server的创建与请求处理可以参见第5章。

Docker Server接收到镜像下载请求之后，通过路由分发最终由具体的处理方法——postImagesCreate来处理。postImagesCreate的源码实现位于./docker/api/server/server.go#L466-L524，其执行流程主要分为3部分：

1）解析HTTP请求中包含的请求参数，包括URL中的查询参数、HTTP Header中的认证信息等；

2）创建镜像下载Job，并为该Job配置环境变量；

3）触发执行镜像下载Job。

9.4.1　解析请求参数

Docker Server接收到Docker Client发送的镜像下载请求之后，首先解析请求参数，并为后续Job的创建与运行提供参数依据。Docker Server解析的请求参数主要有HTTP请求URL中的查询参数“fromImage”、“repo”以及“tag”，还有HTTP请求Header中的“X-Registry-Auth”。

请求参数解析的源码如下：

```
var (
     image = r.Form.Get("fromImage")
     repo  = r.Form.Get("repo")
     tag   = r.Form.Get("tag")
     job   *engine.Job
)
authEncoded := r.Header.Get("X-Registry-Auth")
```

需要特别说明的是，通过“fromImage”解析出的image变量包含镜像repository名称与镜像tag信息。例如，若用户请求为docker pull ubuntu：14.04，则通过“fromImage”解析出的image变量值为ubuntu：14.04，并非只有Docker镜像的名称。

另外，Docker Server通过从HTTP Header中解析出authEncoded，还原出类型为registry.AuthConfig的对象authConfig，源码实现如下：

```
authConfig := &registry.AuthConfig{}
if authEncoded != "" {
          authJson := base64.NewDecoder(base64.URLEncoding, strings.NewReader
(authEncoded))
     if err := json.NewDecoder(authJson).Decode(authConfig); err != nil {
          // for a pull it is not an error if no auth was given
               // to increase compatibility with the existing api it is
defaulting to be empty
          authConfig = &registry.AuthConfig{}
     }
}
```

解析出HTTP请求中的参数之后，Docker Server对于image参数再次进行解析，从中解析出属于镜像的repository与tag信息，其中repository有可能暂时包含Docker Registry信息，源码实现如下：

```
if tag == "" {
     image, tag = parsers.ParseRepositoryTag(image)
}
```

Docker Server的参数解析工作至此全部完成，之后Docker Server将创建镜像下载任务并开始执行。

9.4.2　创建并配置Job

Docker Server只负责接收Docker Client发送的请求，并将其路由分发至相应的处理方法来处理，最终的请求执行还需要Docker Daemon来协作完成。Docker Server在处理方法中，通过创建Job并触发job执行的形式，把控制权交给Docker Daemon。

Docker Server创建镜像下载job并配置环境变量的源码实现如下：

```
job = eng.Job("pull", image, tag)
job.SetenvBool("parallel", version.GreaterThan("1.3"))
job.SetenvJson("metaHeaders", metaHeaders)
job.SetenvJson("authConfig", authConfig)
```

其中，创建的Job名为pull，含义是下载Docker镜像，传入参数为image与tag，配置的环境变量有parallel、metaHeaders与authConfig。

9.4.3　触发执行Job

Docker Server创建完Docker镜像下载Job之后，需要触发该Job执行，实现将控制权交给Docker Daemon。

Docker Server触发执行Job的源码如下：

```
if err := job.Run(); err != nil {
     if !job.Stdout.Used() {
          return err
     }
     sf := utils.NewStreamFormatter(version.GreaterThan("1.0"))
     w.Write(sf.FormatError(err))
}
```

由于Docker Daemon在启动时，已经配置了名为“pull”的Job所对应的处理方法，实际为graph包中的CmdPull函数，故一旦该Job被触发执行，控制权将直接交给Docker Daemon的CmdPull函数。Docker Daemon启动时Engine的处理方法注册可以参见第3章。

9.5　Docker Daemon

Docker Daemon是完成Job执行的主要载体。Docker Server为镜像下载Job准备好所有的参数配置之后，只等Docker Daemon来完成执行，并返回相应的信息，Docker Server再将响应信息返回至Docker Client。Docker Daemon对于镜像下载Job的执行涉及的内容较多：首先解析Job参数，获取Docker镜像的repository、tag以及Docker Registry信息等；随后与Docker Registry建立会话（session）；然后通过会话下载Docker镜像；接着将Docker镜像下载至本地并存储于Graph中；最后在TagStore中标记该镜像。

Docker Daemon对于镜像下载Job的执行主要依靠CmdPull函数。这个CmdPull函数与Docker Client的CmdPull函数完全不同，前者是为了代替用户发送镜像下载的请求至Docker Daemon，而Docker Daemon的CmdPull函数则实现代替用户真正完成镜像下载的任务。调用CmdPull函数的对象类型为TagStore，其源码实现位于./docker/graph/pull.go。

9.5.1　解析Job参数

正如Docker Client与Docker Server一样，Docker Daemon执行镜像下载Job时的第一个步骤也是解析参数。解析工作一方面要确保传入参数无误，另一方面也要按需为Job提供参数依据。表9-1罗列出Docker Daemon解析的Job参数。

表9-1　Docker Daemon解析的Job参数

参数名称	参数描述
localName	代表镜像的 repository 信息，有可能携带 Docker Registry 信息
tag	代表镜像的标签信息，默认为 latest
authConfig	代表用户在指定 Docker Registry 上的认证信息
metaHeaders	代表请求中的 HTTP Header 信息
hostname	代表 Docker Registry 信息，从 localName 解析获得，默认为 Docker Hub 地址
remoteName	代表 Docker 镜像的 repository 名称信息，不携带 Docker Registry 信息
endpoint	代表 Docker Registry 完整的 URL，从 hostname 扩展获得

参数解析过程中，Docker Daemon还添加了一些精妙的设计。如：在TagStore类型中设计了pullingPool对象，用于保存正在下载的Docker镜像，下载完毕之前禁止其他Docker Client发起相同镜像的下载请求，下载完毕之后pullingPool中的该记录被清除。Docker Daemon一旦解析出localName与tag两个参数信息，就立即检测pullingPool，源码实现位于./docker/graph/pull.go#L36-L46，如下：

```
c, err := s.poolAdd("pull", localName+":"+tag)
if err != nil {
     if c != nil {
               // Another pull of the same repository is already taking place;
just wait for it to finish
               job.Stdout.Write(sf.FormatStatus("", "Repository %s already being
pulled by another client. Waiting.", localName))
          <-c
          return engine.StatusOK
     }
     return job.Error(err)
}
defer s.poolRemove("pull", localName+":"+tag)
```

9.5.2　创建session对象

为了下载Docker镜像，Docker Daemon与Docker Registry需要建立通信。为了保障两者之间通信的可靠性，Docker Daemon采用了session机制。Docker Daemon每收到一个Docker Client的镜像下载请求，都会创建一个与之相应的Docker Registry的session，之后所有的网络数据传输都在该session上完成。包registry定义了session，位于./docker/registry/registry.go，如下：

```
type Session struct {
	authConfig    *AuthConfig
	reqFactory    *utils.HTTPRequestFactory
	indexEndpoint string
	jar           *cookiejar.Jar
	timeout       TimeoutType
}
```

CmdPull函数中创建session的源码实现如下：

```
r, err := registry.NewSession(authConfig,
registry.HTTPRequestFactory(metaHeaders), endpoint, true)
```

创建的session对象为r，在执行镜像下载过程中，多数与镜像相关的数据传输均在r这个seesion的基础上完成。

9.5.3　执行镜像下载

Docker Daemon之前所有的操作都属于配置阶段，从解析Job参数，到建立session对象，而并未与Docker Registry建立实际的连接，并且也还未真正传输过有关Docker镜像的内容。

完成所有的配置之后，Docker Daemon进入Docker镜像下载环节，实现Docker镜像的下载，源码位于./docker/graph/pull.go#L69-L71，如下：

```
if err = s.pullRepository(r, job.Stdout, localName, remoteName, tag, sf,
job.GetenvBool("parallel")); err != nil {
     return job.Error(err)
}
```

以上代码中pullRepository函数包含了镜像下载整个流程的林林总总，下载流程如图9-3所示。

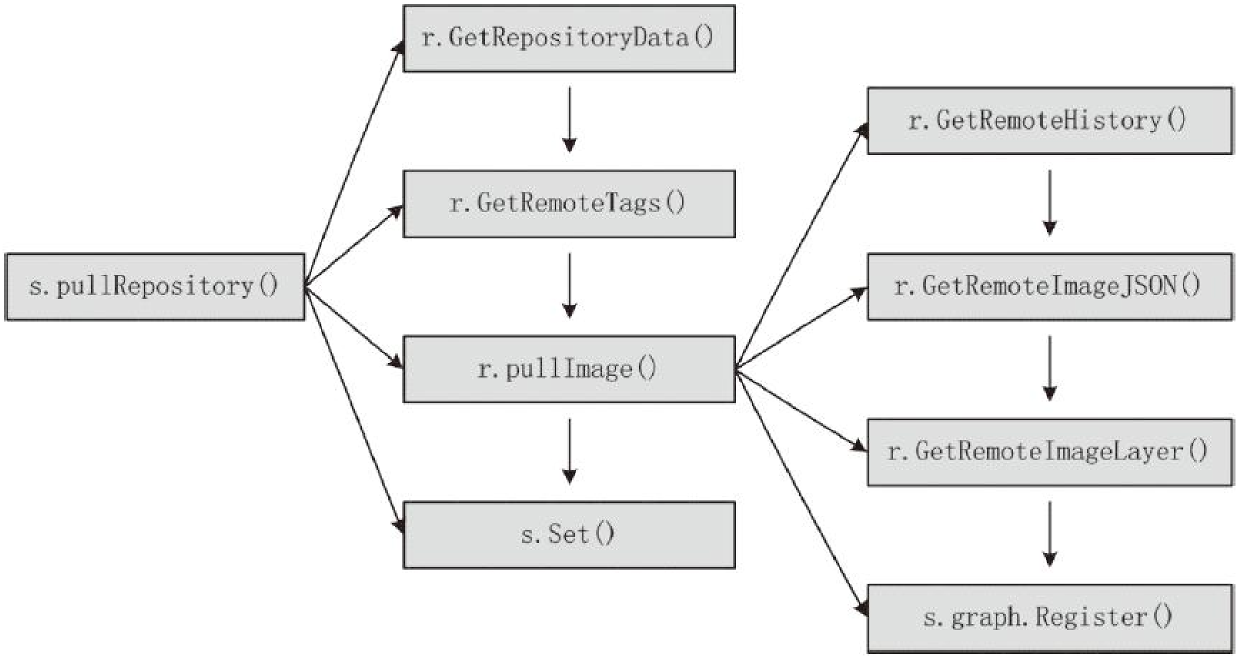


图9-3　pullRepository中镜像下载的流程图

图9-3中各个环节的简要功能介绍见表9-2。

表9-2　pullRepository各环节功能

函数名称	功能介绍
r.GetRepositoryData()	获取指定 repository 中所有镜像的 ID 信息
r.GetRemoteTags()	获取指定 repository 中所有的 tag 信息
r.pullImage()	从 Docker Registry 下载 Docker 镜像
r.GetRemoteHistory()	获取指定 image 所有祖先镜像的 ID 信息
r.GetRemoteImageJSON()	获取指定 image 的 json 信息
r.GetRemoteImageLayer()	获取指定 image 的 layer 信息
s.graph.Register()	将下载的镜像在 TagStore 的 graph 中注册
s.Set()	在 TagStore 中添加新下载的镜像信息

分析pullRepository的整个流程之前，我们有必要了解pullRepository函数调用者的类型TagStore。TagStore是Docker镜像方面涵盖内容最多的数据结构：一方面TagStore管理Docker的

Graph，另一方面TagStore还管理Docker的repository记录。除此之外，TagStore还管理4.6.7节提到的对象pullingPool以及pushingPool，保证Docker Daemon在同一时刻只为一个Docker Client执行同一镜像的下载或上传。TagStore结构体的定义位于./docker/graph/tags.go#L20-L29，如下：

```
type TagStore struct {
     path         string
     graph        *Graph
     Repositories map[string]Repository
     sync.Mutex
     // FIXME: move push/pull-related fields
     // to a helper type
     pullingPool map[string]chan struct{}
     pushingPool map[string]chan struct{}
}
```

下面将重点分析pullRepository的整个流程。

1.GetRepositoryData

使用Docker下载镜像时，用户往往指定的是Docker镜像的名称，如：请求docker pull ubuntu：14.04中镜像名称为ubuntu。GetRepositoryData的作用则是获取镜像名称所在repository中所有image的ID信息。

GetRepositoryData的源码实现位于./docker/registry/session.go#L255-L324。获取repository中image的id信息的目 标URL地址的源码如下：

```
repositoryTarget := fmt.Sprintf("%srepositories/%s/images", indexEp, remote)
```

因此，docker pull ubuntu：14.04请求被执行时，repository的目 标URL地址为https: //index.docker.io/v1/repositories/ubuntu/images，访问该URL可以获得有关ubuntu这个repository中所有image的ID信息，部分image的ID信息如下：


```
[{"checksum": "", "id":
"2427658c75a1e3d0af0e7272317a8abfaee4c15729b6840e3c2fca342fe47bf1"}, {"checksum":
"", "id": "81fbd8fa918a14f4ebad9728df6785c537218279081c7a120d72399d3a5c94a5"},
{"checksum": "", "id":
"ec69e8fd6b0236b67227869b6d6d119f033221dd0f01e0f569518edabef3b72c"}, {"checksum":
"", "id": "9e8dc15b6d327eaac00e37de743865f45bee3e0ae763791a34b61e206dd5222e"},
{"checksum": "", "id":
"78949b1e1cfdcd5db413c300023b178fc4b59c0e417221c0eb2ffbbd1a4725cc"},......]
```


获取以上信息之后，Docker Daemon通过RepositoryData和ImgData类型对象来存储ubuntu这个repository中所有image的信息，RepositoryData和ImgData中数据结构的关系如图9-4所示。

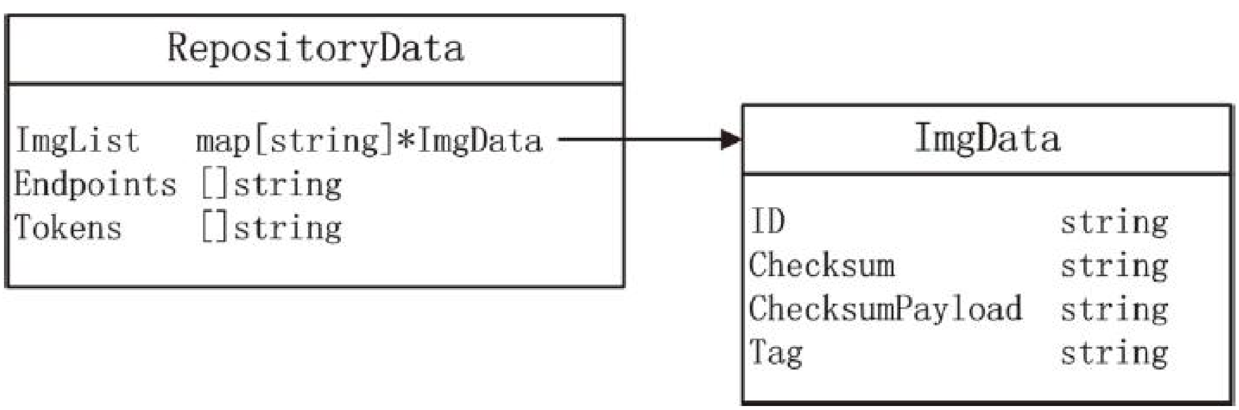


图9-4　RepositoryData和ImgData中数据结构的关系

GetRepositoryData执行过程中，会为指定repository中的每一个image创建一个ImgData对象，并最终将所有ImgData存放在RepositoryData的ImgList属性中，ImgList的类型为map，key为image的ID，value指向ImgData对象。此时ImgData对象中只有属性ID与Checksum有内容。

2.GetRemoteTags

使用Docker下载镜像时，用户除了指定Docker镜像的名称之外，一般还需要指定Docker镜像的tag，如：请求docker pull ubuntu：14.04中镜像名称为ubuntu，镜像tag为14.04，假设用户不显式指定tag，则默认tag为latest。GetRemoteTags的作用则是获取镜像名称所在repository中所有tag的信息。

GetRemoteTags的源码实现位于./docker/registry/session.go#L195-234。获取repository中所有tag信息的目 标URL地址的源码如下：

```
endpoint := fmt.Sprintf("%srepositories/%s/tags", host, repository)
```

获取指定repository中的所有tag信息之后，Docker Daemon根据tag对应layer的ID，找到ImgData，并填充ImgData中的Tag属性。此时，RepositoryData的ImgList属性中，有的ImgData对象中有Tag内容，有的ImgData对象中没有Tag内容。这也和实际情况相

符，如果下载一个ubuntu：14.04镜像，该镜像的rootfs中只有最上层的layer才有tag信息，这一个layer的父镜像并不一定存在tag信息。

3.pullImage

Docker Daemon下载Docker镜像是通过镜像ID来完成的。GetRepositoryData和GetRemoteTags则成功完成了用户传入的repository和tag信息与镜像ID之间的转换。如请求docker pull ubuntu：14.04中，repository为ubuntu，tag为14.04，则对应的镜像ID为2d24f826。需要注意的是，ubuntu：14.04镜像的layer并不是一成不变的，Docker Hub上关于该镜像的内容仍然会更新。

Docker Daemon获得下载镜像的镜像ID之后，首先查验pullingPool，判断是否有其他Docker Client同样发起了该镜像的下载请求，若没有Docker Daemon才继续下载任务。

执行pullImage函数的源码实现位于./docker/graph/pull.go#L159，如下：

```
s.pullImage(r, out, img.ID, ep, repoData.Tokens, sf)
```

而pullImage函数的定义位于./docker/graph/pull.go#L214-L301。图9-3中，我们可以看到pullImage函数的执行可以分为4个步

骤：GetRemoteHistory、GetRemoteImageJSON、GetRemoteImageLayer与s.graph.Register（ ）。

GetRemoteHistory的作用很好理解，既然Docker Daemon已经通过GetRepositoryData和GetRemoteTags找出了指定tag的image id，那么Docker Daemon所需完成的工作为下载该image及其所有的祖先image。GetRemoteHistory负责获取指定image及其所有祖先image的id。

GetRemoteHistory的源码实现位于./docker/registry/session.go#L72-L101。

获取所有的image id之后，对于每一个image id，Docker Daemon都开始下载该image的全部内容。一个layer的Docker Image包括两个方面的内容：image json信息以及image layer信息。Docker所有image的json信息都由函数GetRemoteImageJSON来获取。分析GetRemoteImageJSON之前，有必要阐述清楚什么是Docker Image的json信息。

Docker Image的json信息是一个非常重要的概念。这部分json唯一标志一个image，不仅标志image的id，还标志image所在layer对应的config配置信息。为了理解以上内容，可以举一个例子：docker build。命令docker build用于通过指定的Dockerfile来创建

一个Docker镜像；对于Dockerfile中所有的命令，Docker Daemon都会为其创建一个新的image，如：RUN apt-get update，ENV path=/bin，WORKDIR/home等。对于命令RUN apt-get update，Docker Daemon需要执行apt-get update操作，对应的rootfs上必定会有内容更新，导致新建的image所代表的layer中有新添加的内容。而如ENV path=/bin，WORKDIR/home这样的命令，仅仅配置了一些容器的运行参数，并没有镜像内容的更新，对于这种情况，Docker Daemon同样创建一个新的layer，并且这个新的layer中内容为空，而命令内容会在这层image的json信息中做更新。总之，可以认为Docker的image包含两部分内容：image的json信息、layer内容。当layer内容为空时，image的json信息被更新。第11章将重点分析Docker中dockerbuild的实现。

清楚Docker image的json信息之后，理解GetRemoteImageJSON函数的作用就变得十分容易。执行GetRemoteImageJSON的源码实现位于./docker/graph/pull.go#L243，如下：

```
imgJSON, imgSize, err = r.GetRemoteImageJSON(id, endpoint, token)
```

GetRemoteImageJSON返回的对象imgJSON代表image的json信息，imgSize代表镜像的大小。通过imgJSON对象，Docker

Daemon立即创建一个image对象，创建image对象的源码实现位于./docker/graph/pull.go#L251，如下：

```
img, err = image.NewImgJSON(imgJSON)
```

而NewImgJSON函数位于包image中，函数返回类型为一个Image对象，Image类型的定义如下：

```
type Image struct {
     ID              string              `json:"id"`
     Parent          string              `json:"parent,omitempty"`
     Comment         string              `json:"comment,omitempty"`
     Created         time.Time           `json:"created"`
     Container       string              `json:"container,omitempty"`
     ContainerConfig  runconfig.Config     `json:"container_config,omitempty"`
     DockerVersion   string              `json:"docker_version,omitempty"`
     Author          string              `json:"author,omitempty"`
     Config           *runconfig.Config    `json:"config,omitempty"`
     Architecture    string              `json:"architecture,omitempty"`
     OS              string              `json:"os,omitempty"`
     Size            int64
     graph Graph
}
```

返回img对象，则说明关于该image的所有元数据已经保存完毕，由于还缺少image的layer中包含的内容，因此下一个步骤即为下载镜像layer的内容，调用的函数为GetRemoteImageLayer，函数的源码位于./docker/graph/pull.go#L270，如下：

```
layer, err := r.GetRemoteImageLayer(img.ID, endpoint, token, int64(imgSize))
```

GetRemoteImageLayer函数返回当前image的layer内容。image的layer内容指的是：该image在parent image之上做的文件

系统内容更新，包括文件的增添、删除、修改等。至此，image的json信息以及layer内容均被Docker Daemon获取，这意味着一个完整的image已经下载完毕。下载image完毕之后，并不意味着Docker Daemon关于Docker镜像下载的Job就此结束，Docker Daemon仍然需要对下载的image进行存储管理，以便Docker Daemon在执行其他（如创建容器等）Job时，能够方便地使用这些image。

Docker Daemon在graph中注册image的源码实现位于./docker/graph/pull.go#L283-L285，如下：

```
err = s.graph.Register(imgJSON,utils.ProgressReader(layer, imgSize, out, sf,
false, utils.TruncateID(id), "Downloading"),img)
```

Docker Daemon通过graph存储image是一个很重要的环节。Docker在1.2.0版本中可以通过aufs、DevMapper以及BTRFS来进行image的存储。在Linux 3.18-rc2版本中，OverlayFS已经被合入内容主线，故从Docker 1.4.0版本开始，Docker的image支持OverlayFS的存储方式。

Docker镜像的存储在Docker中是较为独立且重要的内容，第10章将对Docker镜像的存储进行深入分析。

4.配置TagStore

Docker镜像下载完毕之后，Docker Daemon需要在TagStore中指定的repository中添加相应的tag。每当用户查看本地镜像时，都可以从TagStore的repository中查看所有含tag信息的image。

Docker Daemon配置TagStore的源码实现位于./docker/graph/pull.go#L206，如下：

```
if err := s.Set(localName, tag, id, true); err != nil {
     return err
}
```

TagStore类型的Set函数定义位于./docker/graph/tags.go#L174-L205。Set函数的执行流程与简要介绍如图9-5所示。

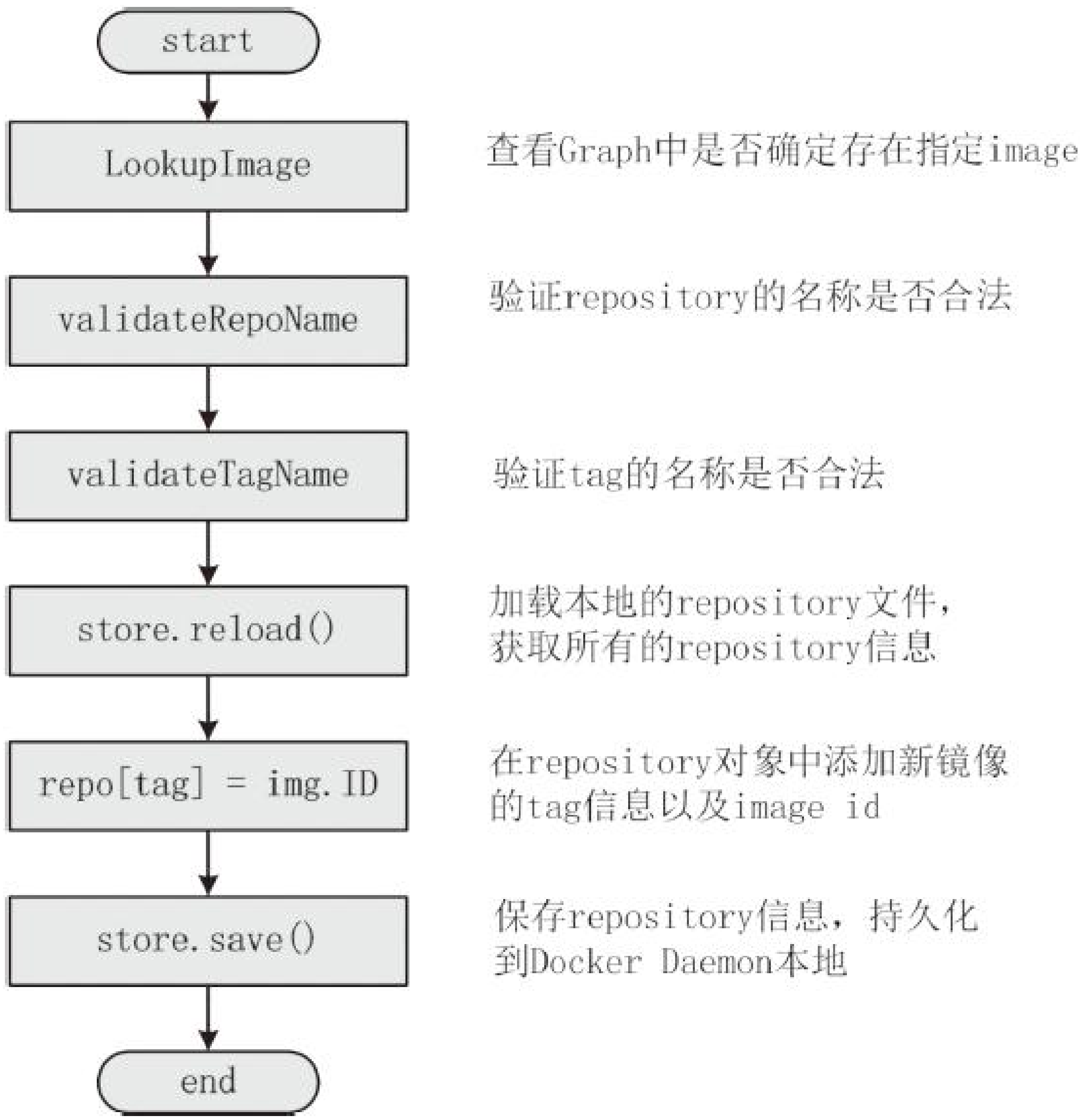


图9-5　TagStore中Set函数的执行流程及简介

当Docker Daemon将已下载的Docker镜像信息同步到repository之后，Docker下载镜像的Job就全部完成，Docker Daemon返回响应至Docker Server，Docker Server返回响应至Docker Client。本地的repository文件位于Docker的根目 录，根目

录一般为/var/lib/docker，如果使用aufs的graphdriver，则repository文件名为repositories-aufs。

9.6　总结

Docker镜像给Docker容器的运行带来了无限的可能性，诸如Docker Hub之类的Docker Registry又使得Docker镜像在全球的开发者之间共享。Docker镜像的下载是使用Docker的第一个步骤。Docker爱好者若能熟练掌握其中的原理，必定能对Docker的很多概念有更为清晰的认识，对Docker容器的运行、管理等均是有百利而无一害。

Docker镜像的下载需要Docker Client、Docker Server、Docker Daemon以及Docker Registry四者协同合作完成。本章从源码的角度分析了四者各自扮演的角色，分析过程中还涉及多种Docker概念，如repository、tag、TagStore、session、image、layer、image json和graph等。

第10章　Docker镜像存储

10.1　引言

Docker Hub汇总众多Docker用户的镜像，极大地发挥Docker镜像开放的思想。Docker用户在全球任意一个位置，都可以与Docker Hub交互，分享自己构建的镜像至Docker Hub，当然，也完全可以下载另一半球Docker开发者上传至Docker Hub的Docker镜像。

无论是上传，还是下载Docker镜像，镜像必然都会以某种形式存储在Docker Daemon所在的宿主机文件系统中。关于Docker镜像在宿主机中的存储，关键点在于：在本地文件系统中以何种组织形式被Docker Daemon有效地统一化管理。这种管理可以使得Docker Daemon创建Docker容器服务时，方便获取镜像并完成Union Mount操作，为容器准备初始化的文件系统。

本章主要从Docker 1.2.0源码的角度，分析Docker Daemon下载镜像过程中存储Docker镜像的环节。本章分析的内容有以下5部分：

1） 概述Docker镜像存储的执行入口，并简要介绍存储流程的四个步骤；

2） 验证镜像ID的有效性；

3） 创建镜像存储路径；

4） 存储镜像内容；

5） 在Graph中注册镜像ID。

10.2　镜像注册

Docker Daemon执行镜像下载任务时，从Docker Registry处下载指定镜像之后，仍需要将镜像合理地存储于宿主机的文件系统中。更为具体而言，存储工作分为两个部分：

1）存储镜像内容；

2）在Graph中注册镜像信息。

提到镜像内容，需要强调的是，每一个layer的DockerImage内容都可以认为由两个部分组成：镜像中每一个layer中存储的文件系统内容，这部分内容一般可以认为是未来Docker容器的静态文件内容；另一部分内容指的是容器的json文件，json文件代表的信息除了容器的基本属性信息之外，还包括未来容器运行时的动态信息，如ENV等信息。

存储镜像内容，意味着Docker Daemon所在宿主机上已经存在镜像的所有内容，除此之外，Docker Daemon仍需要对所存储的镜像进行统计备案，以便用户在后续的镜像管理与使用过程中，可以有据可循。为此，Docker Daemon设计了Graph，使用Graph来接管这部分的工作。Graph负责记录有哪些镜像已经正确存储，供Docker Daemon调用。

Docker Daemon执行CmdPull任务的pullImage阶段时，实现Docker镜像存储与记录的源码位于./docker/graph/pull.go#L283-L285，如下：

```
err = s.graph.Register(imgJSON,utils.ProgressReader(layer, imgSize, out, sf,
false, utils.TruncateID(id), “Downloading”),img)
```

以上源码的实现，实际调用了函数Register，Register函数的定义位于./docker/graph/graph.go#L162-L218，如下：

```
func (graph *Graph) Register(jsonData []byte, layerData archive.ArchiveReader,
img *image.Image) (err error)
```

分析以上Register函数的定义，可以得出以下内容：

1）函数名称为Register；

2）函数调用者类型为Graph；

3）传入函数的参数有3个：第一个为jsonData，类型为数组；第二个为layerData，类型为archive.ArchiveReader；第三个为img，类型为*image.Image；

4）函数返回对象为err，类型为error。

Register函数的运行流程如图10-1所示。

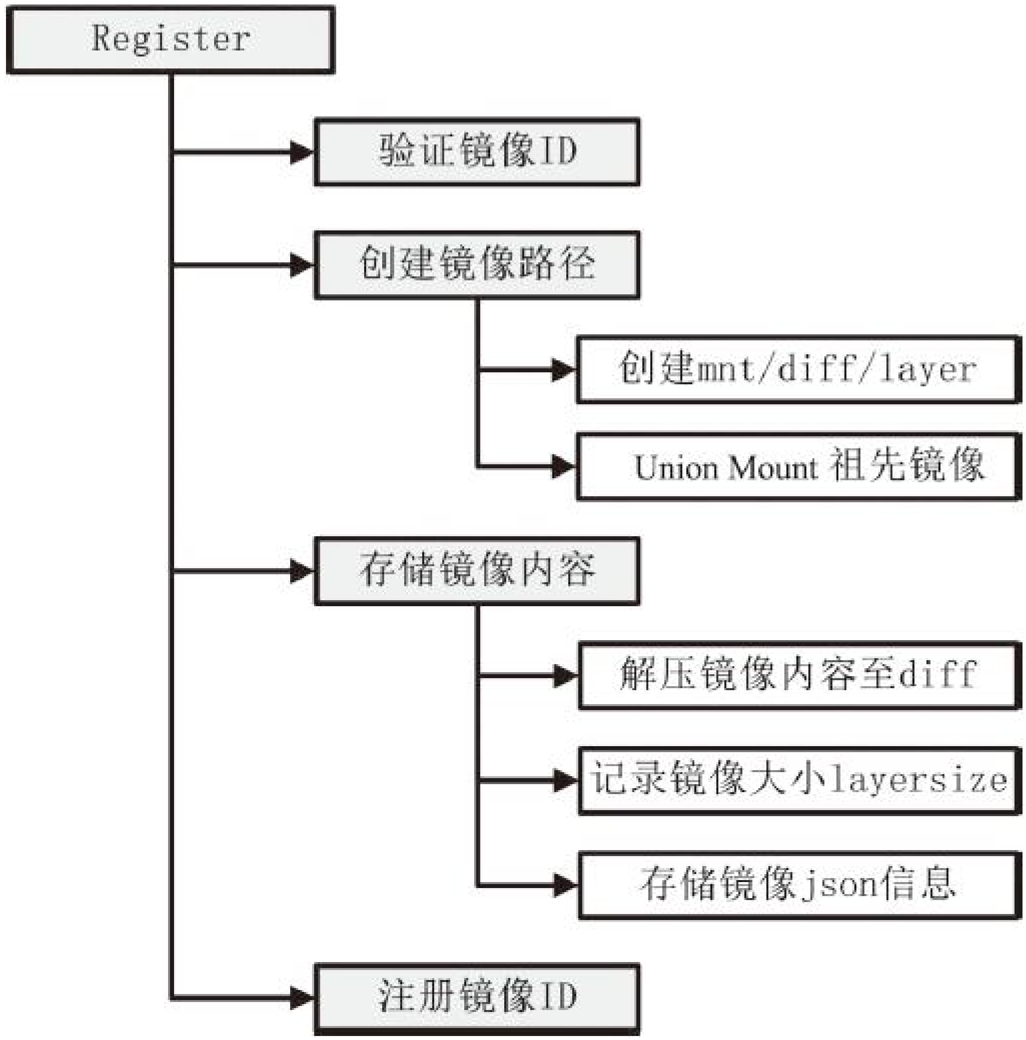


图10-1　Register函数执行流程

10.3　验证镜像ID

Docker镜像注册的第一个步骤是验证Docker镜像的ID。此步骤确保镜像ID命名的合法性。就功能而言，这部分内容提高了Docker镜像存储环节的鲁棒性。验证镜像ID由三个环节组成。

1）验证镜像ID的合法性；

2）验证镜像是否已存在；

3）初始化镜像目录。

验证镜像ID的合法性使用包utils中的ValidateID函数完成，实现源码位于./docker/graph/graph.go#L171-L173，如下：

```
if err := utils.ValidateID(img.ID); err != nil {
     return err
}
```

ValidateID函数的实现过程中，Docker Dameon检验了镜像ID是否为空，以及镜像ID中是否存在字符“：”，以上两种情况只要其中之一成立，Docker Daemon就认为镜像ID不合法，不予执行后续内容。

镜像ID的合法性验证完毕之后，Docker Daemon接着验证镜像是否已经存在于Graph中。若该镜像已经存在于Graph中，则Docker Daemon返回相应错误，不予执行后续内容。代码实现如下：

```
if graph.Exists(img.ID) {
     return fmt.Errorf("Image %s already exists", img.ID)
}
```

验证工作完成之后，Docker Daemon为镜像准备存储路径。该部分源码实现位于./docker/graph/graph.go#L182-L196，如下：

```
if err := os.RemoveAll(graph.ImageRoot(img.ID)); err != nil &&
!os.IsNotExist(err) {
     return err
}
// If the driver has this ID but the graph doesn't, remove it from the driver to
start fresh.
// (the graph is the source of truth).
// Ignore errors, since we don't know if the driver correctly returns
ErrNotExist.
// (FIXME: make that mandatory for drivers).
graph.driver.Remove(img.ID)
tmp, err := graph.Mktemp("")
defer os.RemoveAll(tmp)
if err != nil {
     return fmt.Errorf("Mktemp failed: %s", err)
}
```

Docker Daemon为镜像初始化存储路径，实则首先删除属于新镜像的存储路径，即如果该镜像路径已经在文件系统中存在，立即删除该路径，确保存储镜像时不会出现路径冲突问题；接着还删除graph.driver中的指定内容，即如果该镜像在graph.driver中存在，卸载该镜像在宿主机上的目录，并将该目录完全删除。以AUFS这种

类型的graphdriver为例，镜像内容存放在/var/lib/docker/aufs/diff目录下，而镜像会被挂载至目录/var/lib/docker/aufs/mnt下的指定位置。

至此，验证Docker镜像ID的工作已经完成，并且Docker Daemon已经完成对镜像存储路径的初始化，使得后续Docker镜像存储时存储路径不会冲突，graph.driver对该镜像的挂载也不会冲突。

10.4　创建镜像路径

创建镜像路径，是镜像存储流程中的一个必备环节，这一环节直接让Docker使用者了解以下概念：镜像以何种形式存在于本地文件系统的何处。创建镜像路径完毕之后，Docker Daemon首先将镜像的所有祖先镜像通过aufs文件系统挂载至mnt下的指定点，最终直接返回镜像所在rootfs的路径，以便后续直接在该路径下解压Docker镜像的具体内容（只包含layer内容）。

10.4.1　创建mnt、diff和layers子目录

创建镜像路径的源码实现位于./docker/graph/graph.go#L198-L206，如下：

```
// Create root filesystem in the driver
if err := graph.driver.Create(img.ID, img.Parent); err != nil {
          return fmt.Errorf("Driver %s failed to create image rootfs %s: %s",
graph.driver, img.ID, err)
}
// Mount the root filesystem so we can apply the diff/layer
rootfs, err := graph.driver.Get(img.ID, "")
if err != nil {
          return fmt.Errorf("Driver %s failed to get image rootfs %s: %s",
graph.driver, img.ID, err)
}
```

以上源码中Create函数在创建镜像路径时起到举足轻重的作用。首先分析graph.driver.Create(img.ID，img.Parent) 的具体实现。由于在Docker Daemon启动时，注册了具体的graphdriver，故graph.driver实际的值为具体注册的driver。方便起见，本章内容全部以aufs类型为例，即在graph.driver为aufs的情况下，阐述Docker镜像的存储。在Ubuntu 14.04系统上，Docker Daemon的根目录一般为/var/lib/docker，而aufs类型driver的镜像存储路径一般为/var/lib/docker/aufs。

aufs这种联合文件系统的实现，在合并多个镜像时起到至关重要的作用。首先介绍Docker Daemon如何为镜像创建镜像路径，以便

支持通过aufs来union镜像。aufs模式下，graph.driver.Create（img.ID，img.Parent）的具体源码实现位于./docker/daemon/graphdriver/aufs/aufs.go#L161-L190，如下：

```
// Three folders are created for each id
// mnt, layers, and diff
func (a *Driver) Create(id, parent string) error {
	if err := a.createDirsFor(id); err != nil {
		return err
	}
	// Write the layers metadata
	f, err := os.Create(path.Join(a.rootPath(), "layers", id))
	if err != nil {
		return err
	}
	defer f.Close()
	if parent != "" {
		ids, err := getParentIds(a.rootPath(), parent)
		if err != nil {
			return err
		}
		if _, err := fmt.Fprintln(f, parent); err != nil {
			return err
		}
		for _, i := range ids {
			if _, err := fmt.Fprintln(f, i); err != nil {
				return err
			}
		}
	}
	return nil
}
```

在Create函数的实现过程中，createDirsFor函数在Docker Daemon根目录下的aufs目录/var/lib/docker/aufs中，创建指定的镜像目录。若当前aufs目录下还不存在mnt、diff这两个目录，则会首先创建mnt、diff这两个目录，并在这两个目录下分别创建代表镜像内容的文件夹，文件夹名为镜像ID，文件权限为0755。假设下载镜像的镜像ID为image_ID，则创建完毕之后，文件系统中的文件

为/var/lib/docker/aufs/mnt/image_ID与/var/lib/docker/aufs/diff/image_ID。回到Create函数中，执行完createDirsFor函数之后，随即在aufs目录下创建了layers目录，并在layers目录下创建image_ID文件。

如此一来，在aufs下的三个子目录mnt、diff以及layers中，分别创建了名为镜像名image_ID的文件。继续深入分析之前，我们直接来看Docker对这三个目录mnt、diff以及layers的描述，如图10-2所示。

```
aufs driver directory structure

.
├── layers // Metadata of layers
│   ├── 1
│   ├── 2
│   └── 3
├── diff  // Content of the layer
│   ├── 1  // Contains layers that need to be mounted for the id
│   ├── 2
│   └── 3
└── mnt    // Mount points for the rw layers to be mounted
    ├── 1
    ├── 2
    └── 3
```

图10-2　aufs driver目录结构图

简要分析图10-2，其中，layers、diff以及mnt为目录/var/lib/docker/aufs下的三个子目录，1、2、3是镜像ID，分别代表三个镜像，三个目录下的1均代表同一个镜像ID。其中layers目录下

保留每一个镜像的元数据，这些元数据是这个镜像的祖先镜像ID列表；diff目录下存储每一个镜像所在的layer，具体包含的文件系统内容；mnt目录下每一个文件都是一个镜像ID，代表在该层镜像之上挂载的可读写layer。因此，下载的镜像中与文件系统相关的具体内容，都会存储在diff目录下的某个镜像ID目录下。

再次回到Create函数，此时mnt、diff以及layers三个目录下的镜像ID文件已经创建完毕。下一步需要完成的是：为layers目录下的镜像ID文件填充元数据。元数据内容为该镜像的所有祖先镜像ID列表。填充元数据的流程如下：

1） Docker Daemon首先通过f，err：＝os.Create（ path.Join（ a.rootPath（ ） ，"layers"，id）） 打开layers目录下镜像ID文件；

2） 然后，通过ids，err：＝getParentIds（ a.rootPath（ ） ，parent） 获取父镜像的祖先镜像ID列表ids；

3） 其次，将父镜像ID写入文件f；

4） 最后，将父镜像的祖先镜像ID列表ids写入文件f。

最终的结果是：该镜像的所有祖先镜像的镜像ID信息都写入layers目录下该镜像ID文件中。

10.4.2　挂载祖先镜像并返回根目录

Create函数执行完毕，意味着创建镜像路径并配置镜像元数据完毕，接着Docker Daemon返回镜像的根目录，源码实现如下：

```
rootfs, err := graph.driver.Get(img.ID, "")
```

Get函数看似返回了镜像的根目录rootfs，实则执行了更为重要的内容——挂载祖先镜像文件系统。具体而言，Docker Daemon为当前层的镜像完成所有祖先镜像的Union Mount。挂载完毕之后，当前镜像的read-write层位于/var/lib/docker/aufs/mnt/image_ID。Get函数的具体实现位于./docker/daemon/graphdriver/aufs/aufs.go#L247-L278，如下：

```
func (a *Driver) Get(id, mountLabel string) (string, error) {
     ids, err := getParentIds(a.rootPath(), id)
     if err != nil {
          if !os.IsNotExist(err) {
               return "", err
          }
          ids = []string{}
     }
     // Protect the a.active from concurrent access
     a.Lock()
     defer a.Unlock()
     count := a.active[id]
     // If a dir does not have a parent ( no layers )do not try to mount
     // just return the diff path to the data
     out := path.Join(a.rootPath(), "diff", id)
     if len(ids) > 0 {
          out = path.Join(a.rootPath(), "mnt", id)
          if count == 0 {
               if err := a.mount(id, mountLabel); err != nil {
```

```
                    return "", err
                }
            }
        }
        a.active[id] = count + 1
        return out, nil
    }
```

分析以上Get函数的定义，可以得出以下内容：

1）函数名为Get；

2）函数调用者类型为Driver；

3）传入函数的参数有两个：id与mountlabel；

4）函数返回内容有两部分：string类型的镜像根目录与错误对象error。

清楚Get函数的定义，再来看Get函数的实现。分析Get函数实现时，有三个部分较为关键，分别是Driver实例a的active属性、mount操作以及返回值out。

首先分析Driver实例a的active属性。分析active属性之前，需要追溯到aufs类型的graphdriver中Driver类型的定义以及graphdriver与Graph的关系。两者的关系如图10-3所示。

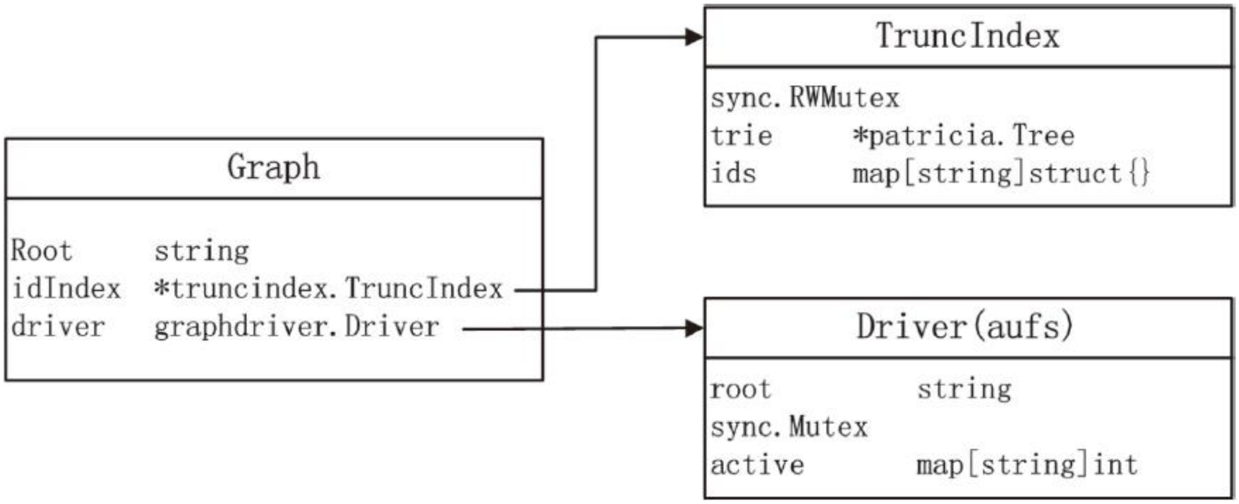


图10-3　Graph与graphdriver的关系

Driver类型的定义位于./docker/daemon/graphdriver/aufs/aufs#L53-L57，如下：

```
type Driver struct {
     root       string
     sync.Mutex // Protects concurrent modification to active
     active     map[string]int
}
```

Driver结构体中root属性代表graphdriver所在的根目录，即/var/lib/docker/aufs。active属性为map类型，key为string，具体运用时key为DockerImage的ID，value为int类型，代表该层镜像layer被引用的次数总和。Docker镜像技术中，某一个layer的Docker镜像被引用一次，则active属性中key为该镜像ID的value值会累加1。用户执行镜像删除操作时，Docker Dameon会检查该Docker镜像的引用次数是否为0，若引用次数为0，则可以彻底删除该镜像，若非零，则仅仅将active属性中引用参数减1。属性

sync.Mutex用于多个Job同时操作active属性时，确保active数据的同步工作。

接着，进入挂载操作的分析。一旦Get参数传入的镜像ID参数不是一个基础镜像，那么说明该镜像存在父镜像，Docker Daemon需要将该镜像所有的祖先镜像都挂载到指定的位置，指定位置为/var/lib/docker/aufs/mnt/image_ID。所有祖先镜像的原生态文件系统内容分别位于/var/lib/docker/aufs/diff/<ID>。其中mount函数用于实现该部分描述的功能，挂载的过程包含很多与aufs文件系统相关的参数配置与系统调用。

最后，Get函数返回out与nil。其中out的值为/var/lib/docker/aufs/mnt/image_ID，即使用该层Docker镜像时其根目录所在路径，也可以认为是镜像的RW层所在路径，但一旦该层镜像之上还有镜像，那么在挂载后者之后，在上层镜像看来，下层镜像仍然是只读文件系统。

10.5 存储镜像内容

存储镜像内容，意味着Docker Daemon以及验证过镜像ID，同时还为镜像准备了存储路径，并返回了其所有祖先镜像执行Union Mount操作后的路径。万事俱备，只欠“镜像内容的存储”。

Docker Daemon为了存储镜像具体内容完成的工作很简单，仅仅是通过某种合适的方式将两部分内容存储于本地文件系统并进行有效管理，两部分内容是镜像压缩内容、镜像json信息。

存储镜像内容的源码实现位于./docker/graph/graph.go#L209-L211，如下：

```
if err := image.StoreImage(img, jsonData, layerData, tmp, rootfs); err != nil {
    return err
}
```

其中，StoreImage函数的定义位于./docker/docker/image/image.go#L74，如下：

```
func StoreImage(img *Image, jsonData []byte, layerData archive.ArchiveReader,
root, layer string) error {
```

分析StoreImage函数的定义，可以得出以下信息：

1）函数名称：StoreImage；

2） 传入函数的参数名：img、jsonData、layerData、root、layer；

3） 函数返回类型error。

传入参数的含义如表10-1所示。

表10-1　StoreImage函数的传入参数

参数名称	参数含义
img	通过下载的 imgJSON 信息创建出的 Image 对象实例
jsonData	Docker Daemon 之前下载的 imgJSON 信息
layerData	镜像作为一个 layer 的压缩包，包含镜像的具体文件内容
root	graphdriver 根目录下创建的临时文件“_tmp”，值为 /var/lib/docker/aufs/_tmp
layer	挂载完所有祖先镜像之后，该镜像在 mnt 目录下的路径

掌握StoreImage函数传入参数的含义之后，理解其实现就十分简单。总体而言，StoreImage亦可以分为三个步骤：

1） 解压镜像内容layerData至diff目录；

2） 收集镜像所占空间大小，并记录；

3） 将jsonData信息写入临时文件。

下面详细介绍三个步骤的实现。

10.5.1　解压镜像内容

StoreImage函数传入的镜像内容是一个压缩包，Docker Daemon理应在镜像存储时将其解压，为后续创建容器时直接使用镜像创造便利。

既然是解压镜像内容，那么这项任务的完成，除了需要代表镜像的压缩包之后，还需要解压任务的目 标路径，以及解压时的参数。压缩包为传入StoreImage的参数layerData，而目 标路径为/var/lib/docker/aufs/diff/< image_ID>。解压流程的源代码位于./docker/docker/image/image.go#L85-L120，如下：

```
// If layerData is not nil, unpack it into the new layer
if layerData != nil {
     if differ, ok := driver.(graphdriver.Differ); ok {
          if err := differ.ApplyDiff(img.ID, layerData); err != nil {
               return err
          }
          if size, err = differ.DiffSize(img.ID); err != nil {
               return err
          }
     } else {
          start := time.Now().UTC()
          log.Debugf("Start untar layer")
          if err := archive.ApplyLayer(layer, layerData); err != nil {
               return err
          }
          log.Debugf("Untar time: %vs", time.Now().UTC().Sub(start).Seconds())
          if img.Parent == "" {
               if size, err = utils.TreeSize(layer); err != nil {
                    return err
               }
          } else {
               parent, err := driver.Get(img.Parent, "")
               if err != nil {
                    return err
               }
               defer driver.Put(img.Parent)
               changes, err := archive.ChangesDirs(layer, parent)
```

```
                if err != nil {
                     return err
                }
                size = archive.ChangesSize(layer, changes)
           }
      }
}
```

可见，当镜像内容layerData不为空时，Docker Daemon需要为镜像压缩包执行解压工作。以aufs这种graphdriver为例，一旦aufs driver实现了graphdriver包中的接口Diff，Docker Daemon就会使用aufs driver的接口方法实现后续的解压操作。解压操作的源代码如下：

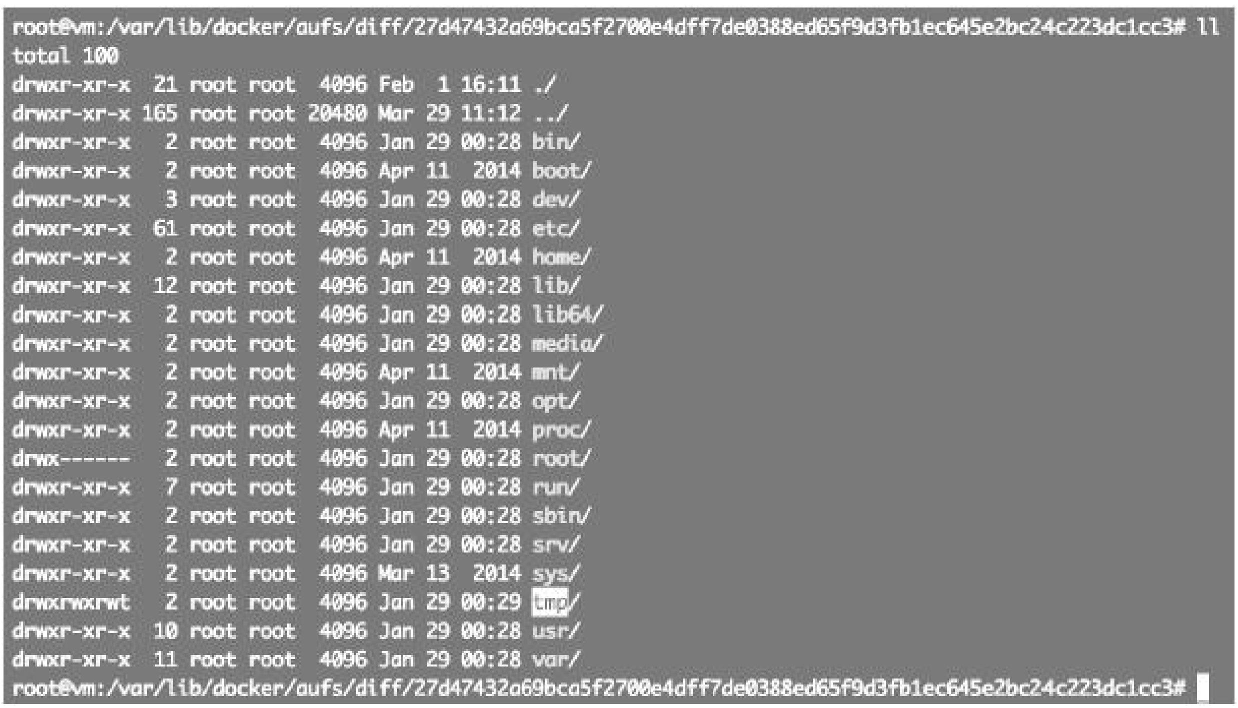
```
if differ, ok := driver.(graphdriver.Differ); ok {
      if err := differ.ApplyDiff(img.ID, layerData); err != nil {
             return err
     }
     if size, err = differ.DiffSize(img.ID); err != nil {
           return err
     }
}
```

以上代码实现了镜像压缩包的解压与镜像所占空间大小的统计。代码differ.ApplyDiff（img.ID，layerData）将layerData解压至目标路径。理清目标路径，且看aufs driver中ApplyDiff的实现，位于./docker/docker/daemon/graphdriver/aufs/aufs.go#L304-L306，如下：

```
func (a *Driver) ApplyDiff(id string, diff archive.ArchiveReader) error {
     return archive.Untar(diff, path.Join(a.rootPath(), "diff", id), nil)
}
```

解压过程中，Docker Daemon通过aufs driver的根目录/var/lib/docker/aufs、diff目录与镜像ID，拼接出镜像的解压路径，并执行解压任务。举例说明diff文件的作用，镜像27d474解压后的内容如图10-4所示。

回到StoreImage函数的执行流中，ApplyDiff任务完成之后，Docker Daemon通过DiffSize开启镜像磁盘空间统计任务。



```
root@vm:/var/lib/docker/aufs/diff/27d47432a69bca5f2700e4dff7de0388ed65f9d3fb1ec645e2bc24c223dc1cc3# ll
total 100
drwxr-xr-x  21 root root  4096 Feb  1 16:11 ./
drwxr-xr-x 165 root root 20480 Mar 29 11:12 ../
drwxr-xr-x   2 root root  4096 Jan 29 00:28 bin/
drwxr-xr-x   2 root root  4096 Apr 11  2014 boot/
drwxr-xr-x   3 root root  4096 Jan 29 00:28 dev/
drwxr-xr-x  61 root root  4096 Jan 29 00:28 etc/
drwxr-xr-x   2 root root  4096 Apr 11  2014 home/
drwxr-xr-x  12 root root  4096 Jan 29 00:28 lib/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 lib64/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 media/
drwxr-xr-x   2 root root  4096 Apr 11  2014 mnt/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 opt/
drwxr-xr-x   2 root root  4096 Apr 11  2014 proc/
drwx------   2 root root  4096 Jan 29 00:28 root/
drwxr-xr-x   7 root root  4096 Jan 29 00:28 run/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 sbin/
drwxr-xr-x   2 root root  4096 Jan 29 00:28 srv/
drwxr-xr-x   2 root root  4096 Mar 13  2014 sys/
drwxrwxrwt   2 root root  4096 Jan 29 00:29 tmp/
drwxr-xr-x  10 root root  4096 Jan 29 00:28 usr/
drwxr-xr-x  11 root root  4096 Jan 29 00:28 var/
root@vm:/var/lib/docker/aufs/diff/27d47432a69bca5f2700e4dff7de0388ed65f9d3fb1ec645e2bc24c223dc1cc3#
```

图10-4　镜像解压后示意图

10.5.2　收集镜像大小并记录

Docker Daemon接管镜像存储之后，Docker镜像被解压到指定路径并非意味着“任务完成”。Docker Daemon还统计了镜像所占磁盘空间的大小，以便记录镜像信息，最终将这类信息传递给Docker用户。

镜像所占磁盘空间大小的统计与记录，实现过程简单且有效，源代码位于./docker/docker/image/image.go#L122-L125，如下：

```
img.Size = size
if err := img.SaveSize(root); err != nil {
     return err
}
```

首先Docker Daemon将镜像大小收集起来，更新Image类型实例img的Size属性，然后通过img.SaveSize(　root)　将镜像大小写入root目录，由于传入的root参数为临时目录_tmp，即写入临时目录_tmp下。实现SaveSize函数的源码如下：

```
func (img *Image) SaveSize(root string) error {
          if err := ioutil.WriteFile(path.Join(root, "layersize"),
[]byte(strconv.Itoa(int(img.Size))), 0600); err != nil {
               return fmt.Errorf("Error storing image size in %s/layersize: %s",
root, err)
     }
     return nil
}
```

SaveSize函数在root目录（临时目录/var/lib/docker/graph/_tmp）下创建文件layersize，并写入镜像大小的值img.Size。

10.5.3　存储jsonData信息

在Docker镜像中，jsonData是一个非常重要的概念。在笔者看来，Docker的镜像并非只是Docker容器文件系统中的文件内容，同时还包括Docker容器运行的动态信息。这里的动态信息更多地是为了适配Dockerfile的标准。以Dockerfile中的ENV参数为例，ENV指定了Docker容器运行时内部进程的环境变量。而这些只有容器运行时才存在的动态信息，并不会被记录在静态的镜像文件系统中，而是以jsonData的形式先存储在宿主机的文件系统中，并与镜像文件系统泾渭分明，存储在不同的位置。当Docker Daemon启动Docker容器时，Docker Daemon会准备好挂载完毕的镜像文件系统环境；接着加载jsonData信息，并在运行Docker容器内部进程时，使用动态的jsonData内部信息为容器内部进程配置环境。

当Docker Daemon下载Docker镜像时，关于每一个镜像的jsonData信息均会被下载至宿主机。通过以上jsonData的功能描述可以发现，这部分信息的存储同样扮演重要的角色。关于Docker Daemon如何存储jsonData信息，实现源码位于./docker/docker/image/image.go#L128-L139，如下：

```
if jsonData != nil {
     if err := ioutil.WriteFile(jsonPath(root), jsonData, 0600); err != nil {
          return err
```

```
        }
    } else {
        if jsonData, err = json.Marshal(img); err != nil {
            return err
        }
        if err := ioutil.WriteFile(jsonPath(root), jsonData, 0600); err != nil {
            return err
        }
    }
```

可见Docker Daemon将jsonData写入了文件jsonPath（root）中，并且把该文件的权限设置为0600。而jsonPath（root）的实现如下，即在root目录（/var/lib/docker/graph/_tmp目录）下创建文件json：

```
func jsonPath(root string) string {
    return path.Join(root, "json")
}
```

镜像大小信息layersize信息统计完毕，jsonData信息也成功记录，两者的存储文件均位于/var/lib/docker/graph/_tmp下，文件名分别为layersize和json。使用临时文件夹来存储这部分信息并非偶然，10.6节将阐述其中的原因。

10.6　注册镜像ID

Docker Daemon执行完镜像的StoreImage操作，回到Register函数之后，执行镜像的commit操作，即完成镜像在Graph中的注册。

注册镜像的代码实现位于./docker/docker/graph/graph.go#L212-L216，如下：

```
// commit
if err := os.Rename(tmp, graph.ImageRoot(img.ID)); err != nil {
        return err
}
graph.idIndex.Add(img.ID)
```

10.5节存储镜像过程中使用到的临时文件_tmp在注册镜像环节有所体现。关于镜像的注册行为，第一步就是将_tmp文件（/var/lib/docker/graph/_tmp）重命名为graph.ImageRoot（img.ID），实则为/var/lib/docker/graph/<img.ID>。使得Docker Daemon在而后的操作中可以通过img.ID在/var/lib/docker/graph目录下搜索到相应镜像的json文件与layersize文件。

成功为json文件与layersize文件配置完正确的路径之后，Docker Daemon执行的最后一个步骤为：添加镜像ID至

graph.idIndex。源代码实现是graph.idIndex.Add(img.ID) ，Graph中idIndex类型为*truncindex.TruncIndex，TruncIndex的定义位于./docker/docker/pkg/truncindex/truncindex.go#L22-L28，如下：

```
// TruncIndex allows the retrieval of string identifiers by any of their unique
prefixes.
// This is used to retrieve image and container IDs by more convenient shorthand
prefixes.
type TruncIndex struct {
     sync.RWMutex
     trie *patricia.Trie
     ids  map[string]struct{}
}
```

Docker用户使用Docker镜像时，一般可以通过指定镜像ID来定位镜像，如Docker官方的mongo：2.6.1镜像ID为c35c0961174d51035d6e374ed9815398b779296b5f0ffceb7613c8199383f4b1，该ID长度为64。当Docker用户指定运行这个Mongo镜像Repository中tag为2.6.1的镜像时，完全可以通过64位的镜像ID来指定，如下：

```
docker run -it c35c0961174d51035d6e374ed9815398b779296b5f0ffceb7613c8199383f4b1
/bin/bash
```

然而，记录如此长的镜像ID，对于Docker用户来说，稍显不切实际，而TruncIndex的概念则极大地帮助Docker用户可以通过简短的ID定位到指定的镜像，使得Docker镜像的使用变得尤为方便。原理是：Docker用户指定镜像ID的前缀，只要前缀满足在全局所有的

镜像ID中唯一，Docker Daemon就可以通过TruncIndex定位到唯一的镜像ID。而graph.idIndex.Add（ img.ID） 正式完成将img.ID添加并保存至TruncIndex中。

为了达到上一条命令的效果，Docker用户完全可以使用TruncIndex的方式，当然，前提是c35这个字符串作为前缀全局唯一，命令如下：

```
docker run –it c35 /bin/bash
```

至此，Docker镜像存储的整个流程已经完成。概括而言，它主要包含验证镜像、存储镜像、注册镜像三个步骤。

10.7　总结

Docker镜像的存储，使得Docker Hub上的镜像能够遍布世界各地变为现实。Docker镜像在Docker Registry中的存储方式与本地化的存储方式并非一致。Docker Daemon必须针对自身的graphdriver类型，选择适配的存储方式，实施镜像的存储。本章也在不断强调一个事实，即Docker镜像并非仅仅包含文件系统中的静态文件，除此之外，它还包含镜像的json信息，json信息中有Docker容器的配置信息，如暴露端口、环境变量等。

可以说Docker容器的运行严重依赖于Docker镜像，因此了解Docker镜像的由来就变得尤为重要。Docker镜像的下载、Docker镜像的打包以及构建新的Docker镜像，都无法跳出镜像存储的范畴。Docker镜像的存储知识，也有助于Docker其他概念的理解，如docker commit、docker build、docker run等。

第11章　docker build实现

11.1　引言

Docker镜像在Docker生态圈中起到的作用非同一般。一起来看Docker的发展，我们甚至可以如此评价：Docker正是凭借着其灵活的镜像技术，以及革命性的镜像托管服务Docker Hub，在云计算时代占 据了行业内极为有利的位置。其自身的Dockerfile技术甚至有可能取代传统软件发布的模式，成为行业内更快捷、更有效、更易于部署与管理的软件发布模式。

Docker诞生之前，软件的发布模式一直以代码为核心。软件开发者首先在开发环境中开发软件，开发到一定阶段后，开发者提交代码至托管平台；软件测试者接着在测试环境中部署软件代码，做相应的测试工作并提交测试报告；软件开发者与测试者将软件完善至一定阶段后交付，交付形式一般仍旧是代码。最终交付的软件，仍然需要在另外的环境中部署，因此软件对部署环境的要求需要通过软件说明书的形式交付给客户。然而，真实的部署环境并不一定能与软件有效地结合，这也是以代码为核心的软件发布模式一直以来的痛点。

Docker诞生之后，软件发布模式似乎有了新的转机。软件代码依然是软件的重要组成部分，然而，与以往不同的是，Docker生态圈中软件的发布使得代码与运行环境并不分离，而是将软件运行环境也圈入软件范畴，一并交付给客户。客户在获得软件之后，并不需要额外地配置环境，而是能够直接运行软件。这种新模式的诞生极大地吸引了开发者的眼球，传统软件发布弊端的剔除，也逐渐使得Docker阵营越来越庞大。

软件发布模式的革新，并非信手拈来，革新理念的外表下是前卫的创新意识，以及创新意识下的技术积淀。Docker的镜像技术，可以说是传统联合文件系统在云时代迎来的又一个春天。在镜像技术之上，Dockerfile的设计则可以认为是Docker公司在镜像技术上抽象出的一个全新软件标准。此标准一出，软件开发者有能力以一个超乎想象的速度，发布包含运行环境的软件，并通过Docker Hub的方式串联全世界。

第8章、第9章以及第10章分别介绍了Docker镜像的原理、Docker镜像的下载以及Docker镜像的存储。在这些镜像技术的基础之上，理解在Docker环境中如何通过Dockerfile构建自己的Docker镜像，就显得意义重大。

本章从Docker 1.2.0的源码出发，分析用户通过Dockerfile构建出一个Docker镜像的来龙去脉。

分析之前，我们首先来简单了解docker build的作用。简单而言，用户可以通过一个自定义的Dockerfile文件以及相关内容，从一个基础镜像起步，对于Dockerfile中的每一条命令，都在原先的镜像layer之上再额外构建一个新的镜像layer，直至构建出用户所需的镜像。

由于docker build命令由Docker用户发起，故docker build的流程会贯穿Docker Client、Docker Server以及Docker Daemon这三个重要的Docker模块。本章也正是以这三个Docker模块为主题，分析docker build命令的执行，其中Docker Daemon作为Docker的重中之重，本章对其的分析也将会更加详细。具体而言，本章内容包含以下3部分：

1）概述docker build命令执行的流程，包含Docker Client、Docker Server以及Docker Daemon；

2）详细分析Docker Daemon对Dockerfile的build实现；

3）简要分析Dockerfile中的命令在Docker Daemon中的执行流程。

11.2　docker build执行流程

docker build可以帮助Docker用户通过Dockerfile的形式构建出自己的Docker镜像。更为细致的描述是：用户通过Docker Client向Docker Server发送一条docker build命令，发送命令时，用户需要首先指定Dockerfile以及相关内容，并通过Docker Client将请求发出；Docker Server接收请求之后，将其路由转发至相应的处理方法；Docker Daemon负责执行请求处理的处理方法，解析Dockerfile并构建出最终的镜像。

Docker Client、Docker Server以及Docker Daemon协同完成build任务的流程图如图11-1所示。

下面几节将从Docker Client、Docker Server以及Docker Daemon三个方面分析docker build的流程。

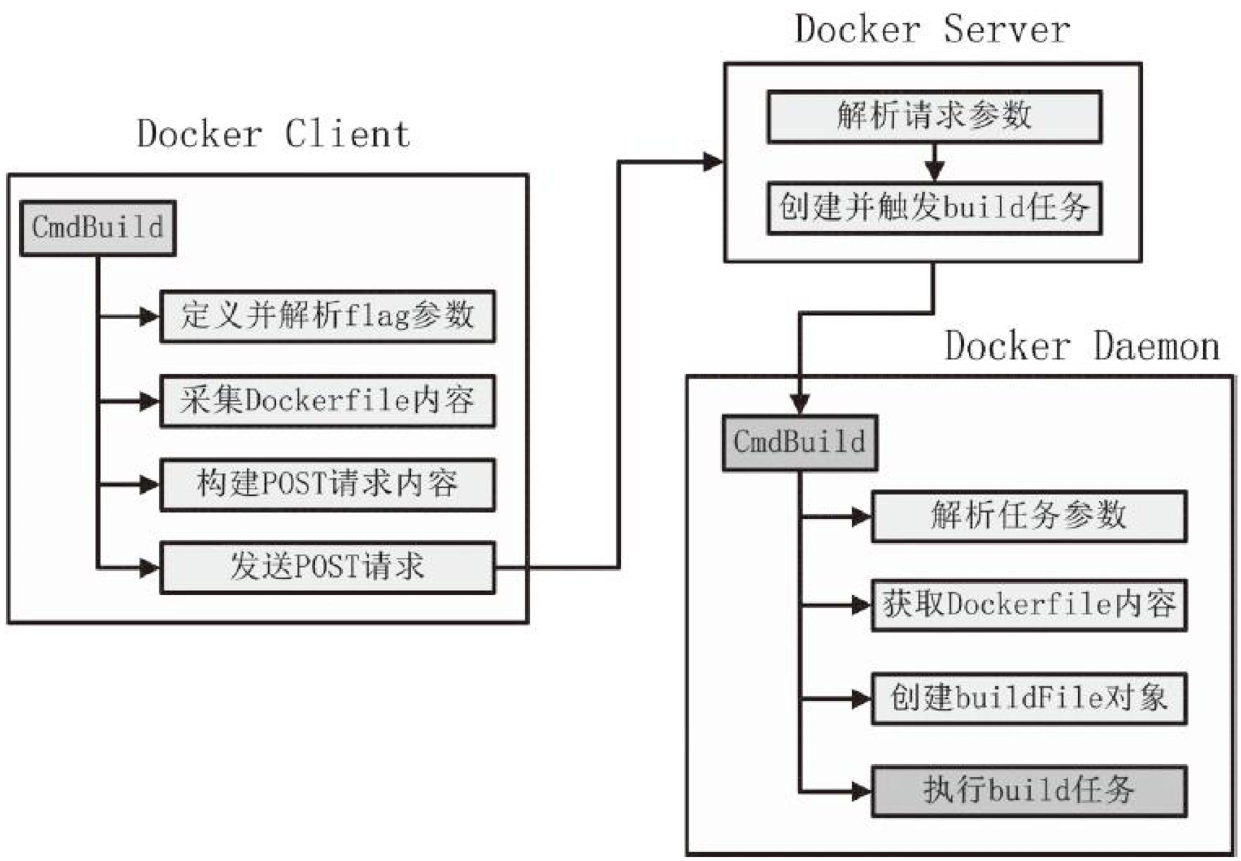


图11-1　docker build的流程图

11.2.1　Docker Client与docker build

Docker Client作为用户请求的入口，自然第一个接收并处理docker build命令。图11-1中也已经清楚地标明Docker Client的处理流程。本章基于Docker 1.2.0版本进行分析，随着Docker版本的不断更新，Docker在build的实现上也有了较为明显的更新，然而，万变不离其宗，docker build的思想与精髓并未产生本质的变化。Docker Client处理docker build命令的流程图如图11-2所示。

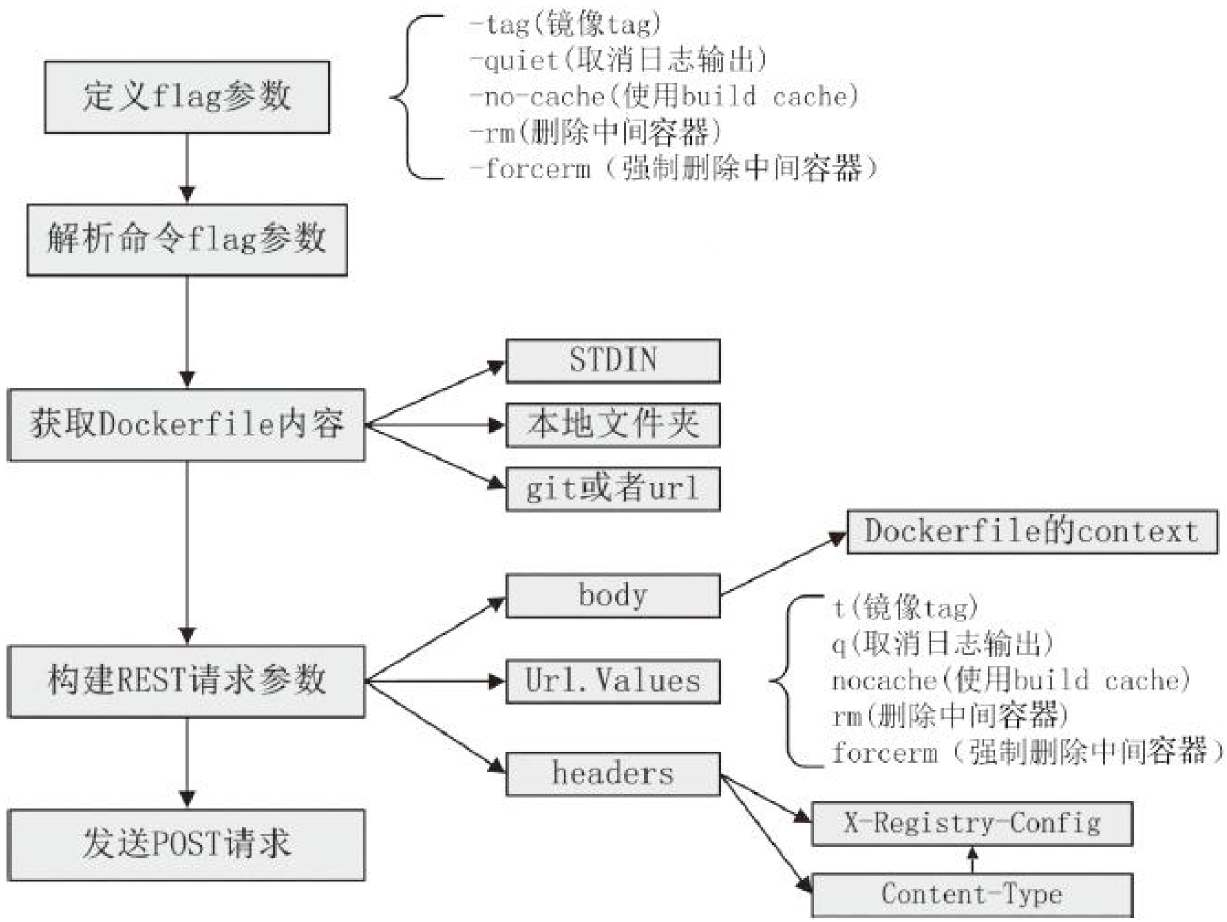


图11-2　Docker Client处理docker build命令的流程图

关于Docker Client端的处理，下面将从4个方面分析docker build命令的处理流程。关于Docker架构中Docker Client的构建与命令执行，可以参考第2章。

1.定义并解析flag参数

用户发起的Docker命令中，很多情况下都会带flag参数。一般情况下，Docker Client会通过定义与解析，对这些flag参数进行转义，从而将用户的请求按需转换，使其适应于Docker Server暴露的API接口。

命令docker build的flag参数共有5个，分别为tag、suppressOutput、noCache、rm与forceRm。flag参数的源码定义位于./docker/docker/api/client/command.go#L102-L106，如下：

```
tag := cmd.String([]string{"t", "-tag"}, "", "Repository name (and optionally a
tag) to be applied to the resulting image in case of success")
suppressOutput := cmd.Bool([]string{"q", "-quiet"}, false, "Suppress the verbose
output generated by the containers")
noCache := cmd.Bool([]string{"#no-cache", "-no-cache"}, false, "Do not use cache
when building the image")
rm := cmd.Bool([]string{"#rm", "-rm"}, true, "Remove intermediate containers after
a successful build")
forceRm := cmd.Bool([]string{"-force-rm"}, false, "Always remove intermediate
containers, even after unsuccessful builds")
```

5个flag参数的具体说明参见表11-1。

表11-1　docker build命令的flag参数说明

flag 参数	flag 参数形式	flag 参数默认值	flag 参数描述
tag	t、-tag	" "（空字符串）	build 完毕后为新镜像设置的 tag 信息
suppressOutput	q、-quiet	false	是否输出容器产生的日志等信息
noCache	-no-cache	false	构建镜像时不使用镜像缓存
rm	-rm	true	成功构建镜像之后，删除中间容器
forceRm	-force-rm	false	强制删除中间容器

需要强调的是：在实际应用场景中，noCache参数的作用非常神奇。一旦-no-cache参数为false，则说明不使用镜像缓存的定论不成立，也就是说，使用镜像缓存。命令docker build命令执行时，镜像缓存会大大缩短相似Dockerfile的build时间。比如：Dockerfile1中第4条命令为编译、安装某一款软件，执行该命令本身需要占用的时间极长；若Dockerfile2中的前4条命令与Dockerfile1完全一致，并且不涉及内容复制，则Dockerfile2在构建前4条命令时，完全可以使用Dockerfile1构建时的镜像缓存，从而大大压缩了本次docker build所消耗的时间。

定义flag参数完毕之后，Docker Client随即解析命令中的flag参数，并对处理结果进行处理，源码实现如下：

```
if err := cmd.Parse(args); err != nil {
       return nil
}
if cmd.NArg() != 1 {
     cmd.Usage()
     return nil
}
```

至此，第一个步骤已经完成，flag参数解析之后，Docker Client将立即进入Dockerfile内容采集阶段。

2.获取Dockerfile相关内容

命令docker build的初衷是为Docker用户构建预期的镜像。为了达到这个目 的，Docker用户自然需要向Docker Daemon提供必需的原材料。而这些原材料正是Dockerfile以及其相关内容。既然如此，Docker Client则必须代表Docker用户提供这些原材料。

Docker Client在flag参数解析完成之后，下一步骤即为获取Dockerfile相关内容。最为常见的Dockerfile一般为一个文件，存在于Docker Client所在的文件系统中，并且伴随着一些其他文件，一般是Dockerfile所在目 录下的其他相关文件内容。用户往往可以在Dockerfile所在目 录下执行docker build命令来完成镜像构建。这是Docker最为普遍的构建方式，分析build的源码实现，大家就可以发现：Docker的build命令支持的Dockerfile源比指定目 录要丰富得多。除此之外，Docker还支持从STDIN读取Dockerfile、从远程URL获取Dockerfile，以及从git源获取Dockerfile。如何解析Dockerfile源并且构建context信息的流程图如图11-3所示。

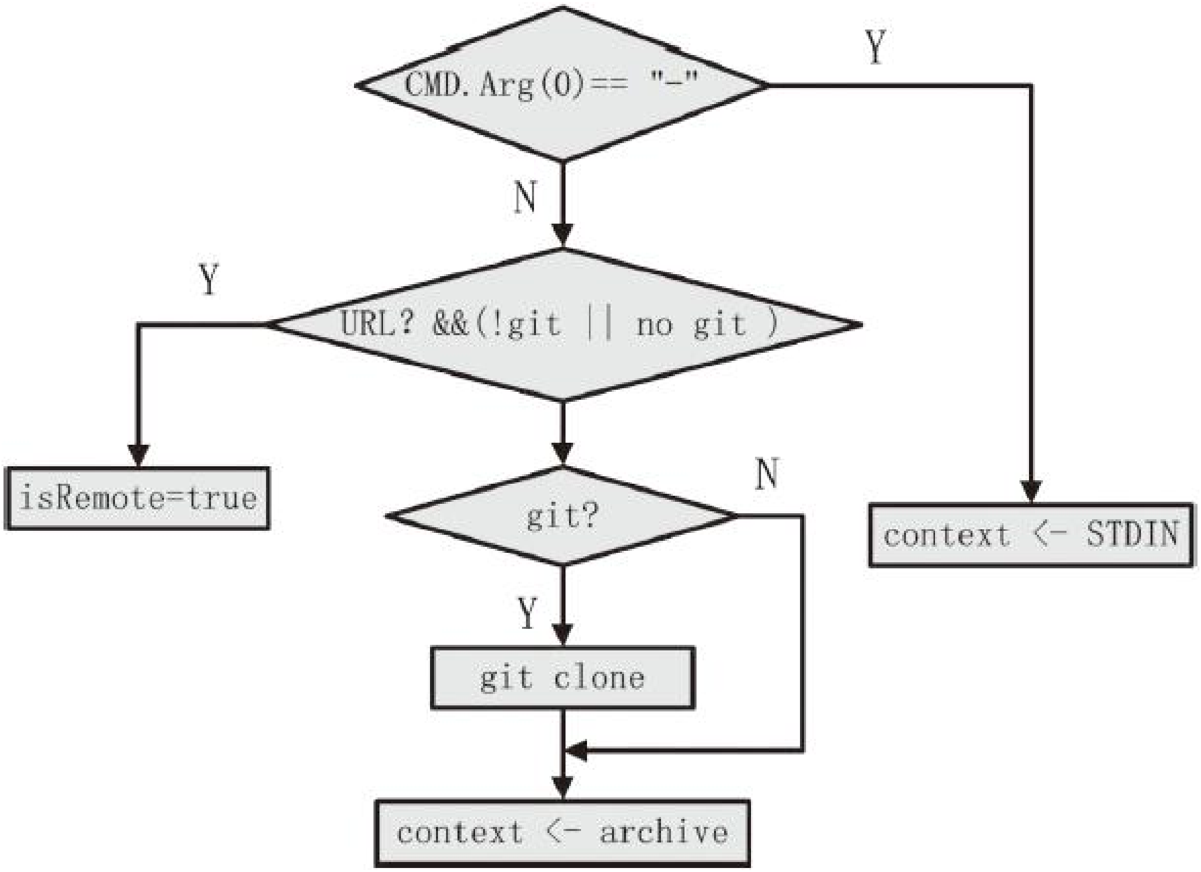


图11-3　解析Dockerfile源并构建context信息的流程图

Docker Client解析Dockerfile源的源码位于./docker/docker/api/client/command.go#L123-L193，如下：

```
if cmd.Arg(0) == "-" {
     ……
     if !archive.IsArchive(magic) {
          dockerfile, err := ioutil.ReadAll(buf)
          if err != nil {
               return fmt.Errorf("failed to read Dockerfile from STDIN: %v", err)
          }
          context, err = archive.Generate("Dockerfile", string(dockerfile))
     } else {
          context = ioutil.NopCloser(buf)
     }
} else if utils.IsURL(cmd.Arg(0)) && (!utils.IsGIT(cmd.Arg(0)) ||  !hasGit) {
     isRemote = true
} else {
     root := cmd.Arg(0)
     if utils.IsGIT(root) {
          ……
               if output, err := exec.Command("git", "clone", "--recursive",
```

```
remoteURL, root).CombinedOutput(); err != nil {
                return fmt.Errorf("Error trying to use git: %s (%s)", err, output)
            }
        }
        ……
        options := &archive.TarOptions{
            Compression: archive.Uncompressed,
            Excludes:    excludes,
        }
        context, err = archive.TarWithOptions(root, options)
        if err != nil {
            return err
        }
}
```

简要分析该流程。Docker Client解析完flag参数之后，通过第一个参数cmd.Arg(0)来确定Dockerfile源。流程大致如下：

1）若该参数为"-"，则代表Docker用户指定STDIN作为Dockerfile的输入源，Docker Client随即通过docker build命令的标准输入读入数据，压缩后最终得到结果context；

2）若该参数与URL匹配，并且不为git地址，又或者参数与URL匹配，同时也是git地址，但是Docker Client所在宿主机没有安装git，则标记isRemote参数为true，表明Dockerfile需要远程获取。

3）若该参数为git地址，且本地安装git，则Docker Client首先将git内容通过git clone命令复制到本地；否则，表明Dockerfile已经位于当前目录，Docker Client解析当前路径之后，将内容压缩打包，得到最终结果context。

Docker Daemon执行docker build命令的原材料已经辨析清楚、准备完毕。此时，Docker Client需要为发送请求给Docker Daemon

做准备。

3.构建REST请求参数

构建REST请求，标志着Docker Client与Docker Daemon建立通信的开始。Docker Client需要为该请求配置相应的参数，比如请求体，请求url中的参数值，以及请求header等。

请求体参数的配置，源码位于./docker/docker/api/cli/command.go#L194-L200，如下：

```
var body io.Reader
// Setup an upload progress bar
// FIXME: ProgressReader shouldn't be this annoying to use
if context != nil {
     sf := utils.NewStreamFormatter(false)
          body = utils.ProgressReader(context, 0, cli.err, sf, true, "", "Sending
            build context to Docker daemon")
}
```

简单而言，请求体的配置依赖context的内容，即Dockerfile以及相关内容将作为请求的body。

请求url的参数配置紧随其后，这些url参数有tag、quiet、remote、nocache、rm与forcerm。除了remote参数的值来源于cmd.Arg(　0)　之外，其他参数均来源于docker build命令的flag参数。

请求参数配置的最后阶段是请求header。由于docker build请求同样有可能需要用户的认证信息，故Docker Client为请求添加了名为X-Registry-Config的header，值为用户的Config信息。同时也会由于请求body的类型，而设置header中的Content-Type参数。

4.发送POST请求

Docker Client万事准备就绪之后，最后一个步骤即发送POST请求至Docker Server，由Docker Server将请求路由至Docker Daemon中具体的处理方法。

发送请求的源码位于./docker/docker/api/cli/command.go#L245，如下：

```
err = cli.stream("POST", fmt.Sprintf("/build?%s", v.Encode()), body, cli.out,
headers)
```

请求类型为POST，请求的URL为/build，并携带url的查询参数值，请求体为body，请求也带有header信息。

当POST请求发送完毕之后，docker build在Docker Client的处理就基本完成了，接着由Docker Server以及Docker Daemon扮演处理请求的角色。

11.2.2 Docker Server与docker build

正如Docker Server一如既往的角色，它负责根据请求类型以及请求的URL，路由转发Docker请求至相应的处理方法。在处理方法中，Docker Server会创建相应的Job，为Job配置相应的执行参数并触发该Job的运行。

命令docker build在Docker Client中最终通过一个POST请求发出，URL前缀为build。而Docker Server的路由表中有以下这条规则，位于./docker/docker/api/server/server/go#L1123，如下：

```
"POST": {
          ...
          "/build":                        postBuild,
          ...
}
```

因此对于docker build请求，Docker Server执行的处理方法为postBuild。postBuild的定义与实现位于./docker/docker/api/server/server.go#L913，如下：

```
func postBuild(eng *engine.Engine, version version.Version, w http.
               ResponseWriter, r *http.Request, vars map[string]string) error
```

此处理方法的职责与其他处理方法无异：解析请求参数，创建并触发执行Job。创建的Job名为build，代码如下：

```
job = eng.Job("build")
```

而Docker Client传递至Docker Server的url参数，均会按规则对job的环境变量赋值。需要特殊说明的是：关于POST请求的body，也就是Dockerfile相关内容，会以Job自身标准输入的形式添加至job内部，代码如下：

```
job.Stdin.Add(r.Body)
```

所有环境变量配置完毕之后，Docker Server执行job.Run，触发执行这个名为build的Job。

11.2.3 Docker Daemon与docker build

Docker Daemon作为Docker体系中的“大脑”部分，任务真正的执行都由它来完成，请求docker build也不例外。处理docker build请求，意味着Docker Daemon会接管build这个Job的执行权，并将任务完成。

Docker Daemon开始执行的任务无外乎也是解析Job环境变量，获取Dockerfile内容。解析Job环境变量的源码位于./docker/docker/daemon/build.go#L40-L53，如下：

```
var (
     remoteURL      = job.Getenv("remote")
     repoName       = job.Getenv("t")
     suppressOutput = job.GetenvBool("q")
     noCache        = job.GetenvBool("nocache")
     rm             = job.GetenvBool("rm")
     forceRm        = job.GetenvBool("forcerm")
     authConfig      = &registry.AuthConfig{}
     configFile      = &registry.ConfigFile{}
     tag            string
     context        io.ReadCloser
)
job.GetenvJson("authConfig", authConfig)
job.GetenvJson("configFile", configFile)
```

Dockerfile内容的获取紧随其后。Dockerfile及其相关内容可以通过三种源向Docker Daemon交付，因此，Docker Daemon获取相关内容时，也需要首先通过参数分辨是哪种具体方式，并实现内容

的最终提取。Docker Daemon获取Dockerfile相关内容context的流程图如图11-4所示。

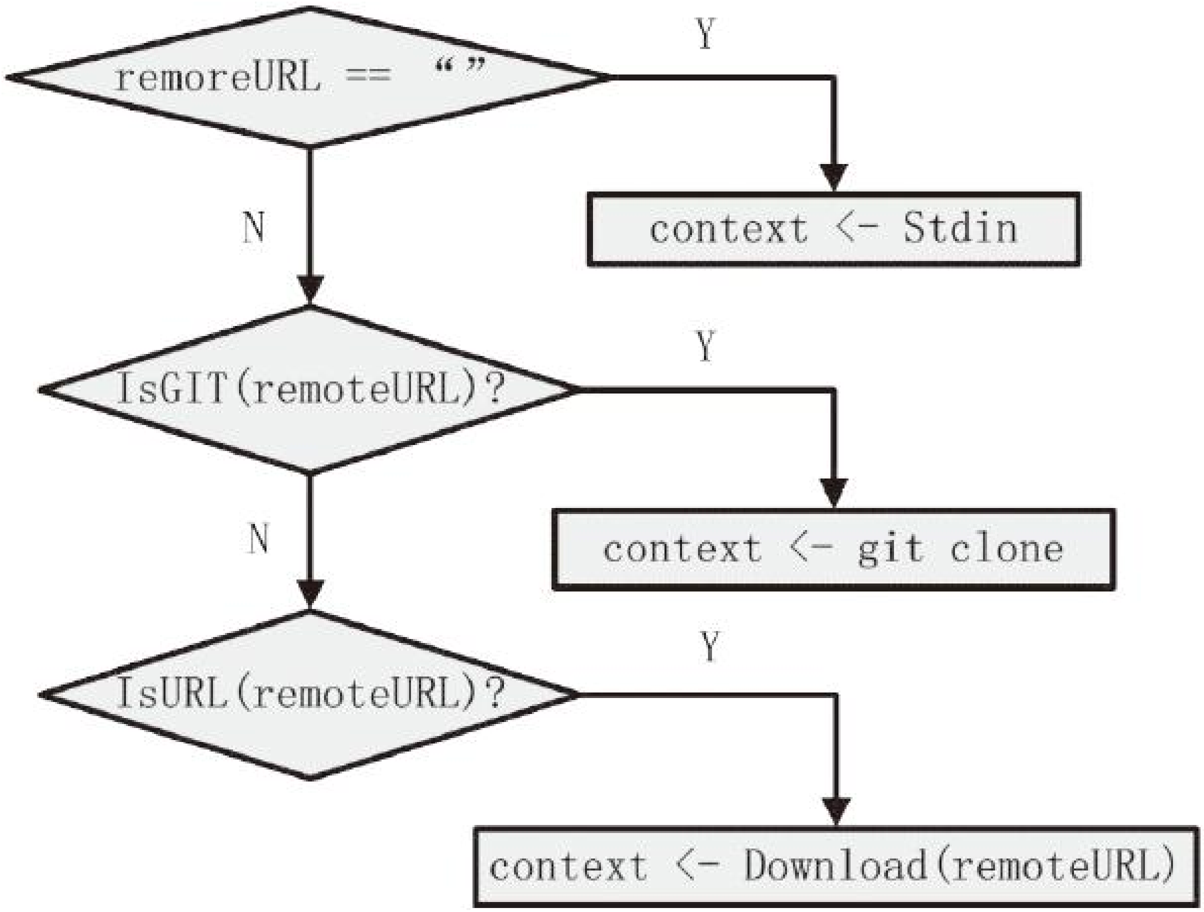


图11-4　Docker Daemon获取context的流程图

这部分的源码实现简单明了，位于./docker/docker/daemon/build.go#L56-L92，如下：

```
if remoteURL == "" {
     context = ioutil.NopCloser(job.Stdin)
} else if utils.IsGIT(remoteURL) {
     if !strings.HasPrefix(remoteURL, "git://") {
          remoteURL = "https://" + remoteURL
     }
     root, err := ioutil.TempDir("", "docker-build-git")
     if err != nil {
          return job.Error(err)
```

```
        }
        defer os.RemoveAll(root)
             if output, err := exec.Command("git", "clone", "--recursive", remoteURL,
                       root).CombinedOutput(); err != nil {
             return job.Errorf("Error trying to use git: %s (%s)", err, output)
        }
        c, err := archive.Tar(root, archive.Uncompressed)
        if err != nil {
             return job.Error(err)
        }
        context = c
   } else if utils.IsURL(remoteURL) {
        f, err := utils.Download(remoteURL)
        if err != nil {
             return job.Error(err)
        }
        defer f.Body.Close()
        dockerFile, err := ioutil.ReadAll(f.Body)
        if err != nil {
             return job.Error(err)
        }
        c, err := archive.Generate("Dockerfile", string(dockerFile))
        if err != nil {
             return job.Error(err)
        }
        context = c
   }
```

分析以上源码，我们可以发现Docker Daemon以下的处理逻辑。若context成功获取，则说明构建Docker镜像的原材料已经准备完毕，Docker Daemon接着将逐一分析context中Dockerfile的内容，并完成相应的build语句。虽然构建Docker镜像的原材料已经完备，但是由Docker Daemon的哪个部件来完成这项重大的工作，仍然未给出答案。紧接着，Docker Daemon根据context获取的源代码，创建一个buildFile对象。命令docker build的生命周期中，buildFile起到的作用非同小可。Dockerfile中书写的所有命令均由buildFile来完成相应的build操作。

首先，我们进入buildFile结构体的定义：

```
type buildFile struct {
	daemon *Daemon
	eng    *engine.Engine
	image      string
	maintainer string
	config     *runconfig.Config
	contextPath string
	context     *tarsum.TarSum
	verbose      bool
	utilizeCache bool
	rm           bool
	forceRm      bool
	authConfig *registry.AuthConfig
	configFile *registry.ConfigFile
	tmpContainers map[string]struct{}
	tmpImages     map[string]struct{}
	outStream io.Writer
	errStream io.Writer
	// Deprecated, original writer used for ImagePull. To be removed.
	outOld io.Writer
	sf     *utils.StreamFormatter
	// cmdSet indicates is CMD was set in current Dockerfile
	cmdSet bool
}
```

对于buildFile结构体，部分属性的含义如表11-2所示。

表11-2　buildFile属性说明

属性名称	含义	属性名称	含义
daemon	Job 所属 Daemon	forceRm	强制删除中间容器
engine	Job 所属 Engine	authConfig	认证信息
image	基础镜像	configFile	配置文件
maintainer	Dockerfile 维护者	tmpContainer	临时容器列表
config	运行配置参数	tmpImages	临时镜像列表
contextPath	context 所在路径	outStream	输出流
context	context 内容	errStream	错误流
verbose	输出 build	outOld	Job 的标准输出
utilizeCache	使用镜像缓存	sf	StreamFormatter
rm	删除中间容器	cmdSet	是否指定 cmd

其中，buildFile可以认为是一个生产镜像车间，只要有原材料（Dockerfile）输入，它就可以按照要求为用户生产Docker镜像。Docker构建并初始化buildFile对象之后，随后即开始真正的build之

旅，车间开始运作，按要求构建镜像。镜像一旦构建成功，Docker Daemon将镜像ID在Repository中注册，以便后续使用。这部分的源码位于./docker/docker/daemon/build.go#L106-L112，如下：

```
id, err := b.Build(context)
if err != nil {
     return job.Error(err)
}
if repoName != "" {
     daemon.Repositories().Set(repoName, tag, id, false)
}
```

构建与注册两项工作的完成，代表Docker Daemon的build任务大功告成。Docker Daemon响应Docker Server，并返回请求响应至Docker Client，通知用户镜像构建任务的完成情况。

Docker Client、Docker Server以及Docker Daemon三者协同完成命令docker build的流程已经分析完毕。目前，我们清晰的是流程，模糊的是具体的build执行。11.3节将给出代码b.Build（context），分析Dockerfile中具体命令的执行细节。

11.3　Dockerfile命令解析流程

Docker Server交付给Docker Daemon的内容是一系列参数，以及Dockerfile相关内容压缩后的context对象，我们习惯于将后者称为原材料。Dockerfile命令解析也是从原材料入手，以下是build函数的执行流程，如图11-5所示。

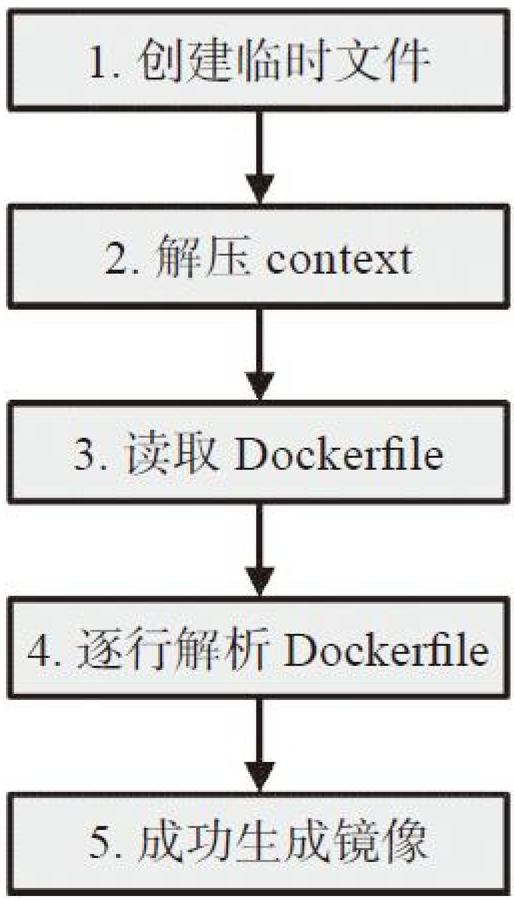


图11-5　build函数执行流程图

函数build的执行过程中，第1～3步，以及第5步的执行内容及意义简单而清晰，本节主要分析第4步——逐行解析Dockerfile。逐行解析Dockerfile命令的代码位于./docker/docker/daemon/build.go#898-L912，如下：

```
for _, line := range strings.Split(dockerfile, "\n") {
     line = strings.Trim(strings.Replace(line, "\t", " ", -1), " \t\r\n")
     if len(line) == 0 {
          continue
     }
     if err := b.BuildStep(fmt.Sprintf("%d", stepN), line); err != nil {
          if b.forceRm {
               b.clearTmp(b.tmpContainers)
          }
          return "", err
     } else if b.rm {
          b.clearTmp(b.tmpContainers)
     }
     stepN += 1
}
```

以上代码的for循环中，每一个循环均会传入Dockerfile中的一行，代码中变量为line。预处理line之后，每次循环执行的任务是b.BuildStep（ ）函数，并在每一个循环的最后，对循环次数进行统计。BuildStep函数的作用是从line中解析相应的Dockerfile指令，完成构建一个镜像layer的任务，并在当前上下文中执行。BuildStep函数的定义位于./docker/docker/daemon/build.go#L921-L943，如下：

```
func (b *buildFile) BuildStep(name, expression string) error {
     fmt.Fprintf(b.outStream, "Step %s : %s\n", name, expression)
     tmp := strings.SplitN(expression, " ", 2)
     if len(tmp) != 2 {
          return fmt.Errorf("Invalid Dockerfile format")
     }
     instruction := strings.ToLower(strings.Trim(tmp[0], " "))
```

```
        arguments := strings.Trim(tmp[1], " ")
             method, exists := reflect.TypeOf(b).MethodByName("Cmd" + strings.
                         ToUpper(instruction[:1]) + strings.ToLower(instruction[1:]))
        if !exists {
                  fmt.Fprintf(b.errStream, "# Skipping unknown instruction %s\n",
                       strings.ToUpper(instruction))
             return nil
        }
             ret := method.Func.Call([]reflect.Value{reflect.ValueOf(b), reflect.
            ValueOf(arguments)})[0].Interface()
        if ret != nil {
             return ret.(error)
        }
        fmt.Fprintf(b.outStream, " ---> %s\n", utils.TruncateID(b.image))
        return nil
}
```

自行编写过Dockerfile的Docker爱好者对于内部每一行的书写肯定非常清楚。Dockerfile的每一行都必须由命令类型和命令参数两部分构成，如：

```
FROM ubuntu:14.04
MAINTAINER Allen Sun allen.sun@daocloud.io
RUN apt-get update
CMD [“/bin/bash”]
```

因此，BuildStep首先通过一行中第一个空格字符""将传入内容分离，如下：

```
tmp := strings.SplitN(expression, " ", 2)
instruction := strings.ToLower(strings.Trim(tmp[0], " "))
arguments := strings.Trim(tmp[1], " ")
```

数组中，tmp[0]代表Dockerfile中相应行的命令类型，tmp[1]代表命令参数。两者解析完毕后，分别赋值给instruction与arguments。命令类型获取之后，Docker Daemon巧妙地使用了Golang中的反射（reflect）机制获取具体的执行方法。执行方法提

取后，执行时传入命令参数，即完成了一条Dockerfile指令的执行，代码如下：

```
method, exists := reflect.TypeOf(b).MethodByName("Cmd" + strings.
                ToUpper(instruction[:1]) + strings.ToLower(instruction[1:]))
ret := method.Func.Call([]reflect.Value{reflect.ValueOf(b), reflect.
                ValueOf(arguments)})[0].Interface()
```

对于Dockerfile内的每一条命令，Docker Daemon都会执行一次循环并通过反射完成方法的执行。如Dockerfile中的命令FROM ubuntu：14.04，首先可以解析出命令类型为FROM，命令参数为ubuntu：14.04，即instruction值为from，arguments为ubuntu：14.04；接着反射机制将通过字符串“CmdFrom”找到方法CmdFrom，即method为CmdFrom；最后通过method.Func.Call（）函数传入参数，完成命令的执行。

11.4节将分析Dockerfile内具体命令的执行。

11.4　Dockerfile命令分析

众所周知，Docker Daemon在构建Dockerfile的过程中，对Dockerfile中的每一条命令（FROM命令除外）都会构建一个新的image。因此，围绕Docker的镜像技术，build的流程也是创建一个个image的过程。

对于一个Dockerfile，或者Dockerfile内的一条或多条命令，buildFile对象均完美地记录所有命令执行时的上下文现状。

Dockerfile内部的命令简单，可以分为两类。第一类命令修改上一层image的文件系统内容，比如：命令RUN在基于上一层image的容器中运行一条指令，对于用户而言，该指令很有可能修改上一层image的内容；命令ADD在Dockerfile所在目录的context中复制内容至上一层image，用户视角下同样属于修改镜像；这样的命令还有COPY等。第二类命令仅仅修改镜像的config信息，比如：命令ENV不会修改镜像文件系统中的内容，而仅仅修改镜像的config的ENV信息，以便后续使用该镜像启动进程时，ENV信息作为进程的环境变量加载；同样EXPOSE命令代表以该镜像运行容器时，容器内进程会监听EXPOSE的端口号，以便Docker Daemon捕获容器内的端口监听情况，这部分信息同样会被记录至config信息，最后被更新至镜像的

json文件中；第二类命令还包括很多，如CMD、ENTRYPOINT、MAINTAINER等。

本节着重分析三种具有代表性的Dockerfile命令FROM、RUN、ENV，阐述这些命令解析执行时，构建Docker新镜像的详细过程。

11.4.1　FROM命令

FROM命令一般都是Dockerfile中的首条命令，紧跟其后的参数为具体的镜像名称，意味着整个build的流程都将此镜像作为基础镜像。虽然作为build流程的基础镜像，但是依旧没有谈及实现方式。此时，buildFile这个对象就体现了重要的作用，很多命令的结果都会在buildFile对象中有所体现。在这里，FROM命令的基础镜像信息会被记录到buildFile对象中，其中与image对象对应的属性值为buildFile.image。

FROM命令的执行流程图如图11-6所示。

从图11-6中，我们可以发现对于FROM命令后的镜像参数，Docker Daemon首先在daemon.repository中查找指定的镜像，若镜像不存在，即立即执行镜像下载任务，Docker镜像的下载可以参见第9章；若镜像存在，则获取镜像信息，并开始buildFile配置流程。配置buildFile的作用是将镜像的信息加载至buildFile。关于FROM命令的解析工作，源码位于

```
b.image = image.ID
b.config = &runconfig.Config{}
if image.Config != nil {
     b.config = image.Config
}
if b.config.Env == nil ||  len(b.config.Env) == 0 {
     b.config.Env = append(b.config.Env, "PATH="+DefaultPathEnv)
}
```

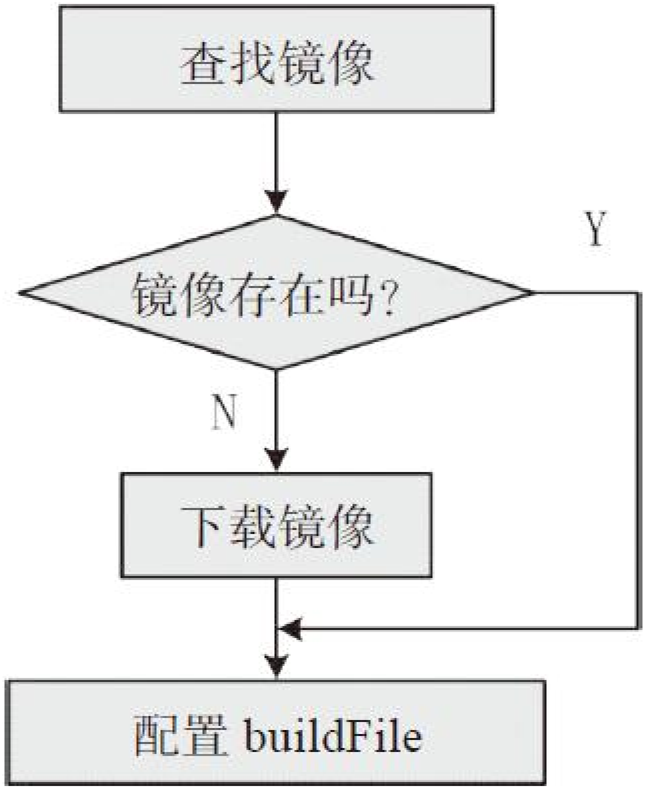


图11-6　FROM命令执行流程图

分析以上源码可以发现，Docker Daemon首先获取镜像ID，将值传给buildFile的image属性。如此一来，build流程的基础镜像信息已经获取，Dockerfile的第二条命令可以在此基础镜像的基础上来完成。当然，buildFile的image属性也会随着build流程的变化而变化。举一个最简单的例子，第三条命令就会依赖第二条命令构建完之后的image。回到buildFile的配置，以上代码表明若镜像config信息存在，则将镜像的config信息传递至buildFile的config属性；而config属性中的Env属性若为空，则在Env属性中添加默认PATH，常量为DefaultPathEnv，如下：

```
const DefaultPathEnv =
"/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
```

CmdFrom执行的最后环节是配置buildFile的OnBuild属性。

总之，CmdFrom完成基础镜像的加载，通过buildFile对象初始化build流程所需的配置信息。

11.4.2　RUN命令

RUN命令在build流程中扮演着非常重要的角色，它允许在镜像的基础上执行用户指定的命令。RUN不同于其他仅修改镜像config信息的命令，前者执行用户指定的命令，这意味着需要在镜像基础上执行动态的命令，执行命令时存在“容器”的概念。

RUN命令的执行通过CmdRun函数来完成，源码实现位于./docker/docker/daemon/build.go#L276-L320，简化后如下：

```
func (b *buildFile) CmdRun(args string) error {
	......
	b.config.Cmd = config.Cmd
	hit, err := b.probeCache()
	if err != nil {
		return err
	}
	if hit {
		return nil
	}
	c, err := b.create()
	......
	c.Mount()
	......
	err = b.run(c)
	......
	if err := b.commit(c.ID, cmd, "run"); err != nil {
		return err
	}
	return nil
}
```

分析以上源码，可以归纳总结出CmdRun函数的执行流程图，如图11-7所示。

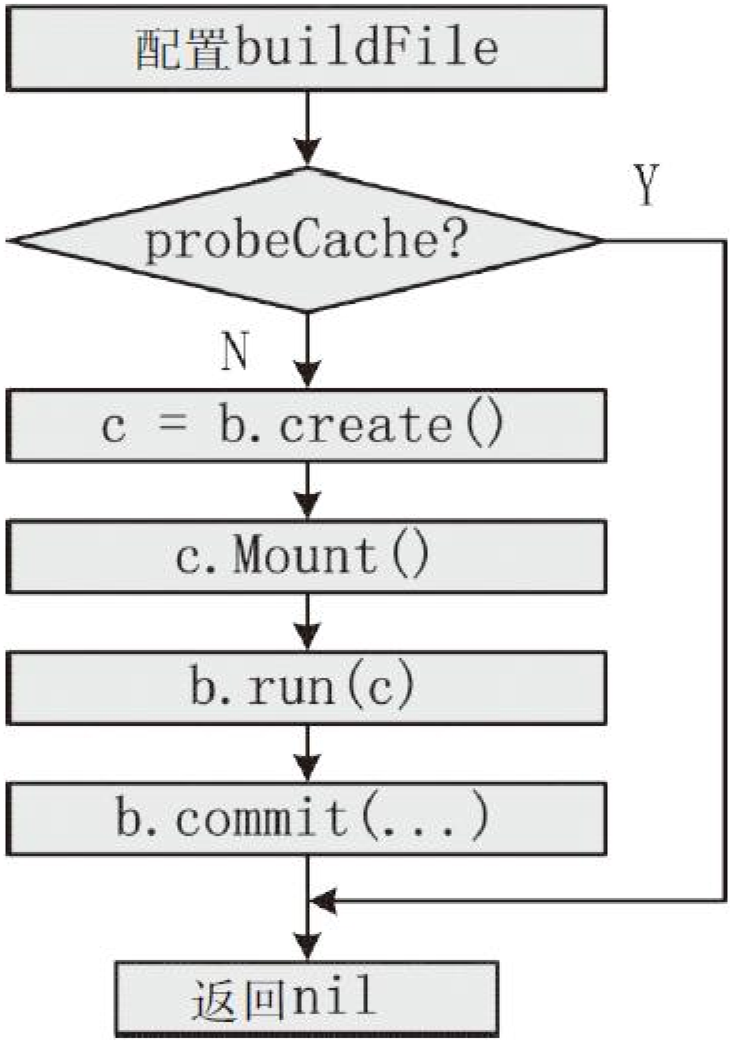


图11-7　CmdRun函数的执行流程图

CmdRun函数的执行流程图，可谓步步精要，每一个步骤在Docker架构中都是一个非常重要的概念。以下从probeCache、b.create（ ）、c.Mount（ ）、b.run（ c）以及b.commit（ ）这5个方面进行分析。

1.镜像cache机制

首先分析probeCache，一起来领略Docker在build流程中巧妙使用镜像cache的精彩。镜像cache的重用意味着构建相应的命令时，不再执行RUN命令后的整条命令，而从Docker Daemon本地的镜像中找出效果与执行该命令相同的镜像，作为结果返回。

实现原理也不难理解。由于Docker Daemon在build RUN命令之前，buildFile的属性image肯定有一个值（基础镜像的ID或者后续覆盖基础镜像的镜像ID）；又由于在执行build RUN命令时，Docker Daemon首先需要配置buildFile的config属性，实则是在buildFile.image的config属性上进行配置。因此，Docker Daemon只需遍历本地所有镜像，只要存在一个镜像，此镜像的父镜像ID与当前buildFile的image值相等，同时此镜像的config内容与buildFile.config相同，则完全可以认为执行RUN命令产生的结果与此镜像的效果一致，直接使用本地存储的此镜像即可，无须额外再创建。若不存在满足条件的镜像，则说明将要构建的新镜像在本地还不存在，Docker Daemon必须完整执行RUN命令。

2.创建Container对象

不能使用镜像Cache机制，意味Docker Daemon没有捷径可走，随后的流程是创建Container对象，为运行容器做准备。

运行Docker容器之前，Docker Daemon会为其创建一个Container对象，RUN命令执行流程中c，err：＝b.create（ ）即创建了一个Container类型的实例c。在Docker Daemon的范畴内，创建一个Container对象，只需要基础镜像的ID，以及运行容器时所需的runconfig信息，而这些信息在build流程中都被buildFile统一管理。函数create的定义位于./docker/docker/daemon/build.go#L752-L771，如下：

```
func (b *buildFile) create() (*Container, error) {
	if b.image == "" {
		return nil, fmt.Errorf("Please provide a source image with 'from'
prior to run")
	}
	b.config.Image = b.image
	// Create the container
	c, _, err := b.daemon.Create(b.config, "")
	if err != nil {
		return nil, err
	}
	b.tmpContainers[c.ID] = struct{}{}
	fmt.Fprintf(b.outStream, " ---> Running in %s\n", utils.TruncateID(c.ID))
	// override the entry point that may have been picked up from the base image
	c.Path = b.config.Cmd[0]
	c.Args = b.config.Cmd[1:]
	return c, nil
}
```

以上代码中，涉及的config参数有当前镜像的ID、所有的buildFile config信息以及config中的Cmd信息（包括Cmd的路径以及Cmd的参数）。

3.挂载文件系统

RUN命令需要在容器中运行指定的程序，故仅仅创建Container类型实例c还不足以支撑容器的运行。CmdRun仍然需要为容器的运行挂载文件系统，c.Mount（ ）即实现了这部分功能。由于Container类型实例c中包含镜像的ID，因此Docker Daemon根据该镜像ID，追溯出此ID的所有祖先镜像，并将所有镜像通过指定的graphdriver完全联合起来，挂载到同一个目录下。而后容器的运行将使用该挂载点，作为容器的根目录。

4.运行容器

运行Docker容器，绝对是Docker体系中的重头戏。如果容器缺少了动态的运行时，对于用户而言，Docker就显得没有丝毫吸引力。

为容器的运行创建了Container类型实例c之后，CmdRun即利用c中众多的容器配置信息，将Docker容器运行起来。在这一过程中，Docker Daemon完成的工作有很多，包括：创建容器的文件系统，创建容器的命名空间进行做资源隔离，为容器配置cgroups参数进行资源控制，当然，还有运行用户指定的程序等。运行容器举足轻重，这部分内容将在第12章详细展开。

5.提交新镜像

运行完容器，DockerDaemon需要对运行后的容器进行commit操作，将容器运行的结果保存在一个新的镜像中，换言之，将更改后

的top layer制作成一个新镜像，并有效存储。需要注意的是，虽然commit操作在执行完容器命令之后，但是如何评价一个命令执行完却有不少难度。一旦RUN命令之后紧跟的命令是无限循环的命令，或者不会退出的命令，容器运行就不会退出，这将导致整个Dockerfile build流程的阻塞。

回到buildFile的commit操作，RUN命令的commit操作和其他命令的commit操作稍有不同。RUN命令的commit操作是从一个运行完毕的容器中保存文件系统中的Read-Write层，以一个镜像的形式存入本地graph中。buildFile的commit函数直接调用daemon包中的Commit函数，源代码位于./docker/docker/build/daemon.go#853，如下：

```
image,err:=b.daemon.Commit(container,"","","",b.maintainer,true,&autoConfig)
```

而daemon包的Commit函数的执行流程包含如下4个步骤。

1）暂停Docker Container的运行。这一步骤对于Dockerfile中的RUN命令不起作用，原因是CmdRun命令只有在命令完全执行完毕，Docker容器终止运行之后，才执行commit操作。然而，对于一个正在运行的容器，Docker同样支持为容器提交一个镜像，而此时Commit操作的第一步即为停止容器的运行，保存容器运行的现场。

2）把容器文件系统的Read-Write层打成tar包。由于在容器的运行过程中所有写操作都只会作用于文件系统中的top layer（即Read-Write层），故容器运行过程中文件系统的增量变化都在Read-Write层，将该层打包即可获得新镜像的文件系统内容。

3）创建image对象，并在Graph中注册。新的tar包即为镜像的原材料，通过tar包，以及众多配置信息，DockerDaemon为其创建一个image对象。最后通过graph.Register（）函数实现在Graph中注册该镜像。graph.Register（）函数相信大家一定不会陌生，第10章已经分析了该函数的实现。

4）在Docker的repositories中注册新创建的镜像。对象repositories的类型实则为TagStore，TagStore的Repositories属性即存储了image的信息，便于用户快速查找。

回到buildFile的CmdRun函数，执行commit操作之后，立即返回创建的新镜像，而将该镜像的ID作为下一个Dockerfile命令执行的基础镜像，源代码为b.image=image.ID，同时CmdRun函数也执行完毕。

11.4.3　ENV命令

ENV命令的含义为：构建镜像时，为镜像添加一个环境变量。ENV命令可以有效帮助用户自定义Docker镜像的环境变量，以至于通过镜像运行容器时，容器进程能拥有用户定义的环境变量。

执行ENV命令的函数为CmdEnv，CmdEnv的主要工作即为在buildFile.image的基础上，配置指定buildFile.config中的Env参数。随后执行commit操作，源码实现如下：

```
b.commit("", b.config.Cmd, fmt.Sprintf("ENV %s", replacedVar))
```

执行以上代码将创建一个新镜像，并运行完daemon包中的Commit函数。虽然创建镜像过程中不会有新的文件系统变化，但是对于镜像而言，镜像的json信息已经发生明显的变化，即镜像的json信息中ENV部分被修改。

11.5 总结

Dockerfile build的流程是不断创建新镜像的过程。有的镜像包含文件系统的内容，这意味着镜像创建过程中，容器的Read-Write层被修改；有的镜像内容为空，即不包含文件系统的内容，这意味着镜像创建过程中，仅仅修改了镜像的json信息。

Dockerfile的存在，使得软件有能力与运行环境一同以一种非常轻量级的方式分发。不论是软件的发布，还是软件的管理方面，Dockerfile都大大释放了软件的生命力。本章即从docker build的流程入手，从原理的角度分析了构建镜像的全过程。熟悉docker build的流程，对于编写高效、合理的Dockerfile，具有非常大的帮助。

第12章　Docker容器创建

12.1　引言

云计算时代，随着Docker的异军突起，工业界刮过阵阵容器的飓风。飓风所到之处，圈内人士纷纷探讨Docker与传统虚拟化技术的异同。传统的虚拟化技术，如KVM、Xen等，在过去的数年间已经受到行业的检验，虽然不是尽善尽美，但给工业界带来的好处也足以让世人对其称赞。Docker的诞生并不是为了替代传统的虚拟化技术，然而，它的诞生却让人如拿着放大镜般地看待传统虚拟化技术的弊病。可以说，Docker指明了又一条虚拟化道路；或许说Docker拓宽了虚拟化的范畴。传统的虚拟化技术主要用于提供基础设施的服务，而Docker在隔离与控制计算资源的同时仍然可以融入用户的应用逻辑，基础设施和上层应用的界限被打破。存在即合理，风靡全球更说明了Docker满足了用户以往那些被认为天方夜谭的需求。

一直认为Docker可以提供“容器”服务，正是Docker提供的“容器”会经常用来与“虚拟机”比较。很多比较的维度大家肯定不陌生，比如：“容器”运行时与宿主机共享同一个操作系统，节省物理资源；“容器”的启动非常快，甚至可以达到秒级，“容器”运行时I/O性能明显高于虚拟机……工业界对两者的比较肯定比以上罗列的更全面、

更具体。然而，似乎很多观点都和“容器”的运行息息相关。而“容器”的运行到底是何种情况？本章将以Docker为例，展现Docker容器运行的始末。

本章主要从源码的角度分析用户发起docker run命令之后，整个Docker体系中的组件如何协同工作，最终实现Docker容器的运行。本书的分析均基于Docker 1.2.0版本，libcontainer的版本也为1.2.0。本章主要内容包含以下三个方面：

1） 简要讲述Docker容器运行的流程，即从Docker Client、Docker Server以及Docker Daemon的角度阐述docker run命令的执行流程；

2） 详细分析Docker Daemon创建容器对象的过程；

3） 详细分析Docker Daemon启动容器的过程。

12.2 Docker容器运行流程

一位精通Docker的好手，对于docker run命令的使用绝对了然于胸。此命令帮助Docker用户在指定配置下完成Docker容器的运行，并运行用户指定的程序。Docker架构中，几乎所有的操作均与docker run有莫大的关联。镜像下拉命令docker pull，目标正是通过docker run命令在镜像的基础上运行容器；镜像构建命令docker build，一旦Dockerfile中涉及RUN命令，每执行一个RUN命令，均会运行一个Docker容器，并对容器进行commit操作，打包容器镜像；而Docker中设置网络模式、容器互联（link）等操作，均需要在容器运行前给docker run命令指定参数。

命令docker run与其他命令无差别，执行流程均由Docker Client、Docker Server以及Docker Server协同完成。第7章分析Docker容器的网络时曾分析过此执行流程，此流程图如图12-1所示（Docker Server仅负责路由转发，图中并未显著标出）：

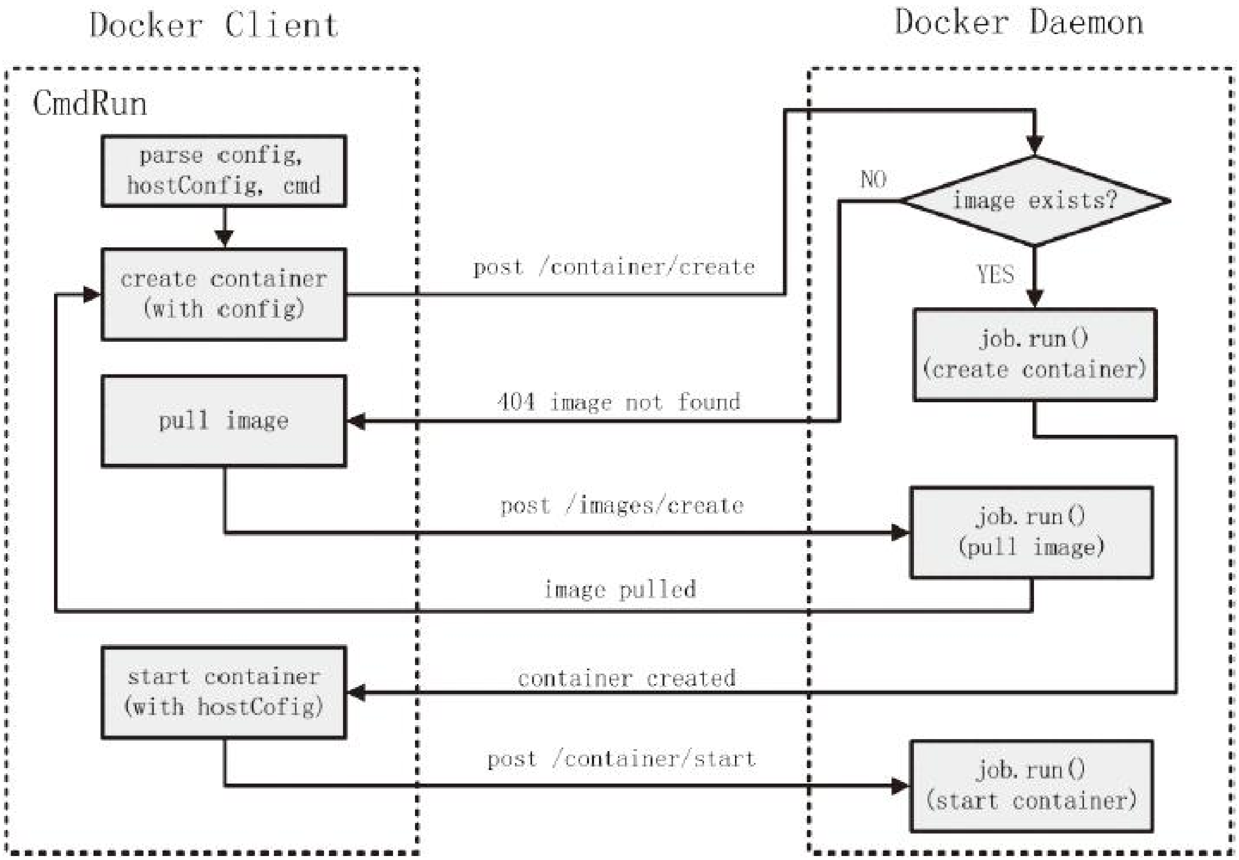


图12-1　docker run命令执行流程图

图12-1表明，Docker Client共发送了两次POST请求至Docker Daemon。第一次请求试图让Docker Daemon通过请求中的镜像信息，创建容器对象；第二次请求则通知Docker Daemon启动容器，使得容器处于运行状态。对于Docker Client而言，docker run命令的所有参数都需要处理成Docker Daemon可以识别的形式，为此Docker的设计者设计了Docker Client与Docker Daemon均可以识别的两个数据结构config与hostconfig，来存储所有命令参数。关于config结构体与hostconfig结构体的描述与分析，可以参见7.3.2节。

为了阐述清楚config与hostconfig中包含的参数信息，本章使用以下docker run命令进行举例分析：

```
docker run -m 104857600 -P --net=bridge -v /home/test:/home/test -v /data -e
TEST_ENV=myenv --name ubuntu_test --link db:db ubuntu:14.04
```

以上命令中参数的分析如表12-1所示。

表12-1　docker run命令参数解析

参数名称	参数形式	参数含义
-m	-m 104857600	为容器运行设置内存上限 104857600 字节，即 100MB
-P	-P	将容器内 EXPOSE 的端口均暴露至宿主机
--net	--net=bridge	容器运行时使用 bridge（桥接）网络模式
-v	-v /home/test:/home/test	将宿主机目录 /home/test 挂载至容器的 /home/test
-v	/data	为容器创建一个非绑定挂载（bind-mount）的 data volume /data
-e	–e TEST_ENV=myenv	为容器添加环境变量 TEST_ENV，值为 myenv
--name	--name ubuntu_test	为容器设置名称 ubuntu_test
--link	--link db:db	将容器 db 以别名 db 的形式链接到创建的容器中

Docker Client的参数解析工作是：处理类似以上的参数并存入config或者hostconfig，最后分别通过两次POST请求发送至Docker Daemon。Docker Client与Docker Server解析处理请求的分析，本章已经提及过，故不再赘述。本章仅从Docker Daemon两次处理并响应POST请求入手，还原Docker容器运行的现场。

12.3 Docker Daemon创建容器对象

Docker容器的运行并非只是简单地调用内核接口，Docker的设计者有意将配置信息与启动容器的操作进行区分，实现数据与逻辑的有机分离。

Docker用户指定的众多参数，以及Docker镜像中的众多参数，在运行容器时，均可以认为是Docker容器的配置信息来源。当第一次接收到Docker Client发送的POST请求后，Docker Daemon开始创建容器对象container，处理并整理用户指定与镜像指定的config信息。创建container对象的源码实现位于./docker/docker/daemon/create.go#L56-L86，如下：

```
// Create creates a new container from the given configuration with a given name.
func (daemon *Daemon) Create(config *runconfig.Config, name string) (*Container,
[]string, error) {
	var (
		container *Container
		warnings  []string
	)
	img, err := daemon.repositories.LookupImage(config.Image)
	if err != nil {
		return nil, nil, err
	}
	if err := img.CheckDepth(); err != nil {
		return nil, nil, err
	}
	if warnings, err = daemon.mergeAndVerifyConfig(config, img); err != nil {
		return nil, nil, err
	}
	if container, err = daemon.newContainer(name, config, img); err != nil {
		return nil, nil, err
	}
	if err := daemon.createRootfs(container, img); err != nil {
		return nil, nil, err
	}
	if err := container.ToDisk(); err != nil {
		return nil, nil, err
```

```
        }
        if err := daemon.Register(container); err != nil {
                return nil, nil, err
        }
        return container, warnings, nil
}
```

Create函数的逻辑极其清晰，从起初声明container对象，到最后返回container，中间环节的逻辑完全属于顺序执行。表12-2给出了中间各环节执行的函数与作用。

表12-2 Create函数执行步骤解析

执行函数名称	执行函数作用
LookupImage	在 daemon 对象的 repositories 属性中查找用户指定镜像
CheckDepth	检验镜像的 layer 总数，镜像层总数不能超过 127
mergeAndVerifyConfig	将用户指定的 config 参数与镜像 json 文件中的 config 合并并验证
newContainer	创建新的 container 对象
createRootfs	创建属于 container 对象的 rootfs
ToDisk	将 container 对象 json 化之后写入本地磁盘进行持久化
Register	在 Docker Daemon 中注册该新建的 container 对象

12.3.1　LookupImage

创建并且运行一个Docker容器，同样需要原材料，这样的原材料是：Docker镜像以及用户配置的信息config对象。而LookupImage函数的功能即为：通过用户指定的镜像名称，从daemon的repositories对象中查找镜像。Docker镜像的概念已经在Docker Daemon的多个概念中出现过，如对象daemon.repositories的类型为TagStore，以及类型Graph与类型Driver等。这三种结构体之间的关系如图12-2所示：

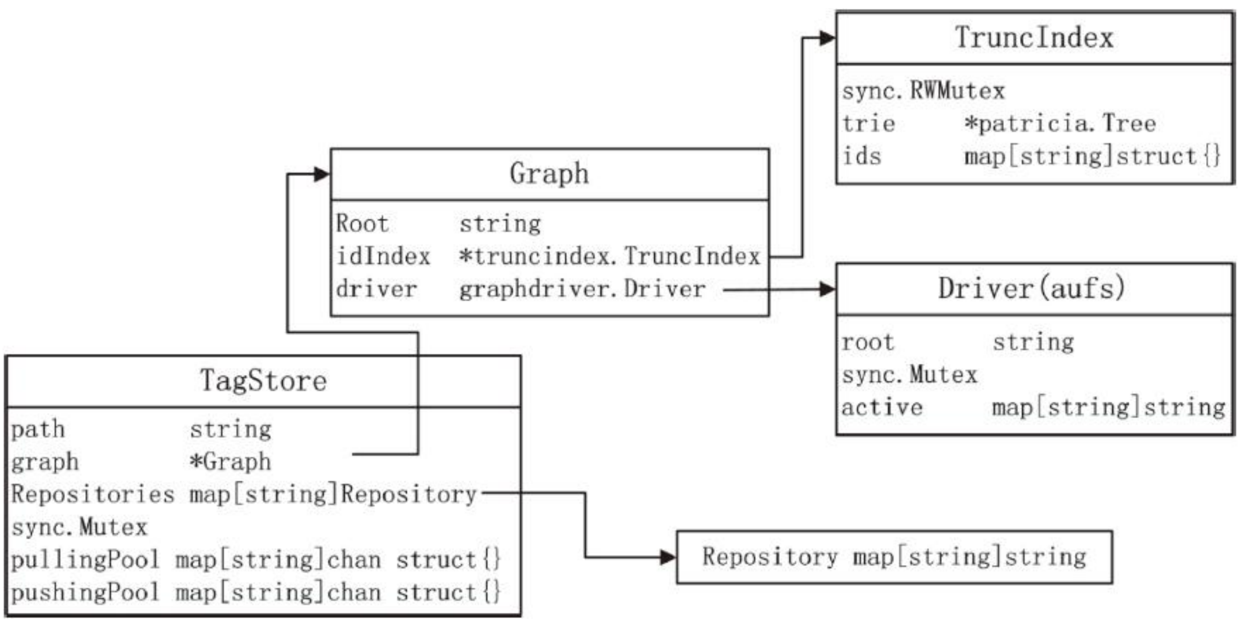


图12-2　TagStore、Graph以及Driver之间的关系

Docker Daemon每次成功下载一个含有tag的镜像，或者每次成功构建一个镜像，最后的步骤都是将这个镜像在

daemon.Repositories中注册，若graghdriver的类型为aufs，则镜像注册会写入本地文件/var/lib/docker/repositories-aufs文件中。文件repositories的存在，使得LookupImage函数的实现变得异常简单，LookupImage函数的源码实现位于./docker/docker/graph/tags.go#L79-L97，如下：

```
func (store *TagStore) LookupImage(name string) (*image.Image, error) {
		// FIXME: standardize on returning nil when the image doesn't exist, and
err for everything else
	// (so we can pass all errors here)
	repos, tag := parsers.ParseRepositoryTag(name)
	if tag == "" {
		tag = DEFAULTTAG
	}
	img, err := store.GetImage(repos, tag)
	store.Lock()
	defer store.Unlock()
	if err != nil {
		return nil, err
	} else if img == nil {
		if img, err = store.graph.Get(name); err != nil {
			return nil, err
		}
	}
	return img, nil
}
```

Docker Daemon需要用户指定镜像名称，而LookupImage函数第一步即解析用户指定镜像的镜像名称与镜像tag，若tag为空，则默认设为“latest”。紧接着，Docker Daemon通过repos(镜像名称)与tag信息，尝试从TagStore的graph属性中提取img对象。Docker Daemon完全通过以上步骤得到img对象，从而获取在本地文件系统中的镜像layer，以及该镜像json文件中的配置信息。

12.3.2 CheckDepth

Docker Daemon找到对应的镜像之后，需要对镜像进行简单的验证。若用户指定的Docker镜像加上所有祖先镜像之后，镜像layer总数达到很大数量，则通过aufs对所有layer执行Union Mount操作将会造成不小的性能问题。最大的问题就是：系统实现文件读写时，查找文件inode的时间变长。因此，Docker Daemon会对用户指定镜像的深度进行检验，源码位于./docker/docker/image/image.go#L308-L319，如下：

```
func (img *Image) CheckDepth() error {

     // We add 2 layers to the depth because the container's rw and // init layer
add to the restriction depth, err := img.Depth() if err != nil {

          return err }

     if depth+2 >= MaxImageDepth {

               return fmt.Errorf("Cannot create container with more than %d
parents", MaxImageDepth) }

     return nil

}
```

Docker Daemon启动容器前，需要通过镜像为容器准备rootfs，由于通过aufs联合layer时，会在镜像的top layer之上，再叠加两个layer，分别为init layer和容器的read-write layer。故CheckDepth函数计算出指定镜像的深度之后，确保容器镜像的深度值加2仍小于最大镜像深度MaxImageDepth。MaxImageDepth的值为常量127。

12.3.3 mergeAndVerifyConfig

分析Docker的镜像技术时，有一点内容经常容易被忽略，那就是：Docker镜像不仅包含文件系统中静态的文件内容，还包含该镜像的json文件信息。而json文件包含镜像的众多描述信息，包括镜像的ID、镜像提交时的容器ID，以及镜像详尽的Config配置信息。

Docker镜像含有config信息，docker run命令传入的许多参数信息也是以runconfig.Config的形式转交至Docker Daemon。

两份Config信息如何在运行容器时各自起到相应的作用，是Docker Daemon应该考虑的问题。而mergeAndVerifyConfig函数则巧妙地将两者有机结合在一起，实现函数为runconfig包中的Merge函数。函数mergeAndVerifyConfig的定义位于./docker/docker/dameon/daemon.go#L399-L413，如下：

```
func (daemon *Daemon) mergeAndVerifyConfig(config *runconfig.Config, img
*image.Image) ([]string, error) {
     warnings := []string{}
          if daemon.checkDeprecatedExpose(img.Config) ||  daemon.check-
DeprecatedExpose(config) {
               warnings = append(warnings, "The mapping to public ports on your
host via Dockerfile EXPOSE (host:port:port) has been deprecated. Use -p to
publish the ports.")
}
     if img.Config != nil {
          if err := runconfig.Merge(config, img.Config); err != nil {
               return nil, err
          }
     }
     if len(config.Entrypoint) == 0 && len(config.Cmd) == 0 {
          return nil, fmt.Errorf("No command specified")
```

```
        }
        return warnings, nil
    }
```

函数mergeAndVerifyConfig除了merge之外，自然还包括Verify和Config，若合并之后的config对象中不存在Entrypoint并且也没有Cmd，则说明整个容器没有启动入口，Docker Daemon必须对这种情况返回错误。

12.3.4 newContainer

Docker Daemon创建容器对象的第4个步骤是：使用newContainer函数新建并初始化container对象。newContainer函数的实现过程很清晰，创建并初始化newContainer函数内部的container对象的源码位于./docker/docker/daemon/daemon.go#L529-L548，如下：

```
container := &Container{

     // FIXME: we should generate the ID here instead of receiving it as an
argument ID:              id,

     Created:         time.Now().UTC(), Path:            entrypoint, Args:
args, //FIXME: de-duplicate from config Config:           config,

     hostConfig:       &runconfig.HostConfig{}, Image:           img.ID, //
Always use the resolved image id NetworkSettings: &NetworkSettings{}, Name:
name,

     Driver:          daemon.driver.String(), ExecDriver:
daemon.execDriver.Name(), State:           NewState(), }

container.root = daemon.containerRoot(container.ID) if container.ProcessLabel,
container.MountLabel, err = label.GenLabels(""); err != nil {

     return nil, err

}
```

其中需要特别说明的是Path属性，Path属性的值为entrypoint。众所周知，entrypoint是Docker容器运行时非常重要的概念。用户一旦指定entrypoint，则Docker运行容器时，可以首先执行entrypoint的内容(一般为脚本)，而entrypoint的最后一行脚本命令一般是exec用户指定的Cmd，届时才开始运行用户指定的应用进程。这样的好处很明显，启动容器时的工作分为两部分：第一，运行entrypoint，完成与系统配置或初始化相关的工作；第二，运行Cmd，开始执行用户应用程序。

12.3.5 createRootfs

完成以上四个步骤，原则上已经成功创建了container对象。此时Docker Daemon并未直接返回container对象，而是在此基础上完成容器文件系统rootfs的配置。文件系统rootfs的挂载，其实要比第8章涵盖的内容更多。通过用户指定的镜像，Docker Daemon完全有能力将所有layer通过aufs联合挂载起来，而createRootfs的实现则是在联合挂载所有镜像layer的基础上，再挂载两个layer，一层称之为“init layer”，另一层则为大家熟知的“read-write layer”。

在createRootfs函数中，创建init layer以及read-write layer的源码实现位于./docker/docker/daemon/daemon.go#L568-L574，如下：

```
if err := graph.SetupInitLayer(initPath); err != nil {
     return err
}
if err := daemon.driver.Create(container.ID, initID); err != nil {
     return err
}
```

graph包的SetupInitLayer的作用是：在镜像基础上挂载一系列与镜像无关而与容器运行环境相关的目录和文件，如/dev/pts、proc、.dockerinit、/etc/hosts、etc/hostname以及/etc/resolv.conf等。需要特殊说明的是：.dockerinit为

dockerinit二进制文件挂载点，而dockerinit是Docker容器中第一个运行的内容。第13章将详细分析dockerinit在Docker容器中的作用。

12.3.6　ToDisk

Docker Daemon对于每一个创建的容器，均会将其json化后持久化至本地文件系统中。ToDisk函数的功能正是如此。ToDisk依靠toDisk函数来完成，后者的源代码定义位于./docker/docker/daemon/container.go#L112-L129，如下：

```
func (container *Container) toDisk() error {

    data, err := json.Marshal(container) if err != nil {

        return err }

    pth, err := container.jsonPath() if err != nil {

        return err }

    err = ioutil.WriteFile(pth, data, 0666) if err != nil {

        return err }

    return container.WriteHostConfig() }
```

以上代码清晰明了，它不仅实现container对象json化后的本地持久化，还实现container中hostConfig对象的本地持久化，最为重

要的自然是确定写入的路径。代码pth，err：＝container.jsonPath（）即获取json文件的放置目录/var/lib/docker/container/containers/<container_id>。

12.3.7　Register

container对象创建之后，并根据一些规则对其进行预处理之后，Docker Daemon将container对象注册至daemon的containers属性。注册的内容为container对象的容器ID。注册的意义在于，Docker Daemon此后可以通过daemon对象调度并管理这个有效的container对象。

Register函数执行完毕，意味着Docker Daemon创建容器对象的工作全部完成。如此一来，万事俱备，只欠东风，Docker Daemon的容器启动命令一触即发。

12.4 Docker Daemon启动容器

Docker的世界中，动态的内容往往更加迷人。静态的内容，诸如镜像、Dockerfile、Docker Registry等，似乎都是为了服务于动态的容器。12.3节中的container对象的创建，也仅仅是静态内容的梳理与组织，仍缺少容器方面的活力。

Docker Daemon启动容器的操作取决于docker run命令发起的第一个POST请求。由于container对象已经创建，并由daemon对象管理，故POST请求只要携带container的ID信息，容器启动命令就能触发。

从逻辑的角度而言，Docker Daemon仅仅通过获取container对象，并执行此对象的Start函数。从Start函数的具体执行内容而言，涉及的内容很广泛，图12-3详细说明Docker Daemon针对container对象所操作的具体内容。

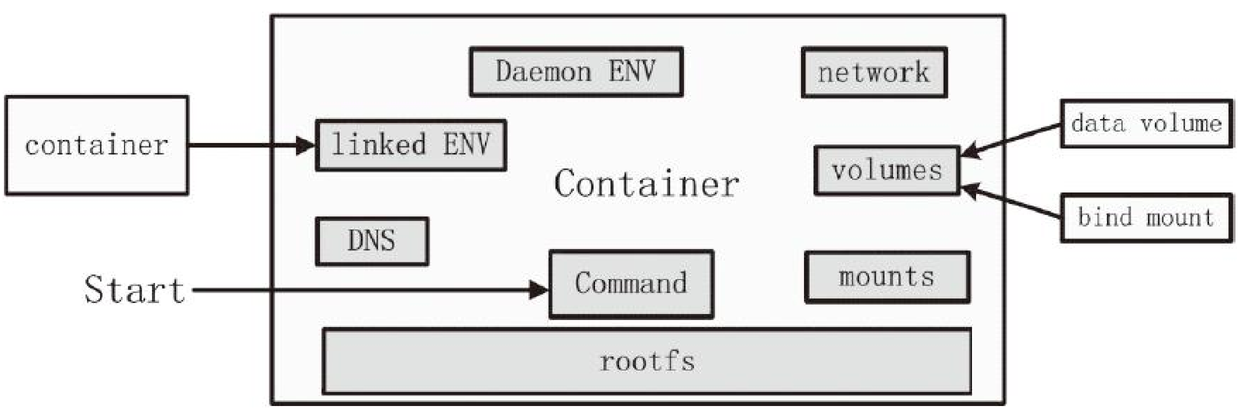


图12-3　Start函数涉及内容示意图

就顺序而言，Docker Daemon执行Start函数，包含以下11个步骤。

12.4.1 setupContainerDns

一个功能完善的Docker容器，肯定离不开基本的网络能力。出色的网络能力意味着容器内部不是一个闭塞的环境，容器可以与外界建立通信。这样的网络自然离不开DNS服务，用于可以确保域名的正常工作。Docker Daemon启动容器的第一个步骤就是配置容器的DNS服务。

一旦Docker容器的网络模式为host模式，则容器和宿主机共享同一个网络命名空间，容器无须为自身的DNS服务考虑，完全使用宿主机的DNS服务。如若不然，Docker Daemon则需要为容器内部配置DNS服务。配置容器DNS的源码实现位于./docker/daemon/container.go#L842-L979，如下：

```
func (container *Container) setupContainerDns() error {
     resolvConf, err := resolvconf.Get()
     ...
     container.ResolvConfPath, err = container.getRootResourcePath("resolv.conf")
     ……
          if config.NetworkMode != "host" && (len(config.Dns) > 0 ||
len(daemon.config.
            Dns) > 0 || len(config.DnsSearch) > 0 || len(daemon.config.DnsSearch)
> 0) {
          var (
               dns       = resolvconf.GetNameservers(resolvConf)
               dnsSearch = resolvconf.GetSearchDomains(resolvConf)
          )
          if len(config.Dns) > 0 {
               dns = config.Dns
          } else if len(daemon.config.Dns) > 0 {
               dns = daemon.config.Dns
          }
          ...
          return resolvconf.Build(container.ResolvConfPath, dns, dnsSearch)
     }
```

```
        return ioutil.WriteFile(container.ResolvConfPath, resolvConf, 0644)
    }
```

创建容器的DNS服务与宿主机有着密不可分的关系。Docker Daemon首先通过resolvconf.Get（）获取宿主机在/etc/resolv.conf文件中的DNS信息；随后在/var/lib/docker/containers/<container_id>/目录下获取resolv.conf文件的路径（此文件最终会被挂载至容器内部）；若容器的网络模式不为host模式，则紧接着的工作是Docker Daemon判断用户和Docker Daemon是否指定DNS地址，一旦指定，则使用指定的DNS地址。

正如上面提及的，Docker用户启动Docker Daemon和启动Docker容器时都可以指定DNS地址。当用户使用docker run命令启动容器时，若使用-dns参数，则表明用户需要为容器指定DNS地址（在Docker Daemon中对应的数据结构为对象config.Dns），Docker Daemon优先满足Docker用户指定DNS的需求。若用户没有指定-dns参数，则Docker Daemon会为容器配置Docker Daemon的DNS地址。如果用户在启动Docker Daemon时指定-dns参数（在Docker Daemon中对应的数据结构为daemon.config.Dns），则表明默认情况下，Docker Daemon会为Docker容器配置该DNS地址。当用户启动Docker Daemon时，若没有指定-dns参数，并且宿主机/etc/resolv.conf文件中第一条记

录的DNS地址为127.0.0.1或者127.0.1.1（容器内部使用这两个DNS地址无效），则Docker Daemon采用默认的DNS地址8.8.8.8和8.8.4.4。采用默认DNS地址并不会一劳永逸。特殊情况下，8.8.8.8和8.8.4.4并不能提供稳定可靠的域名解析服务，很大程度上影响Docker容器的使用。若用户没有指定任何-dns参数，且宿主机的resolv.conf中第一条DNS记录地址不为127.0.0.1和127.0.1.1，则Docker Daemon会为容器配置宿主机的DNS地址。

Docker容器DNS地址的问题，不仅仅存在于docker run命令中，Dockerfile的build过程中只要涉及RUN命令，就也有可能牵扯到DNS问题，故不论是启动Docker Daemon时的-dns参数还是启动Docker容器时的-dns参数，均不可小觑。

12.4.2 Mount

容器的运行离不开文件系统的支持，而找到容器的根目录则被视为一项基本工作。Docker环境中，容器的文件系统概念要比平常复杂一些。文件系统在容器看来是一个树型结构，但是对于宿主机乃至Docker Daemon而言却并非如此。

Mount（）函数即实现获取容器的根目录，并赋予container.basefs，container.Mount（）的实现通过daemon的Mount函数来完成，源码实现位于./docker/docker/daemon/daemon.go#L907-L919，如下：

```
func (daemon *Daemon) Mount(container *Container) error {

     dir, err := daemon.driver.Get(container.ID, container.GetMountLabel())

     if err != nil {

               return fmt.Errorf("Error getting container %s from driver %s: %s",
container.ID, daemon.driver, err)

     }

     if container.basefs == "" {

          container.basefs = dir
```

```
    } else if container.basefs != dir {

            return fmt.Errorf("Error: driver %s is returning inconsistent
paths for container %s ('%s' then '%s')",daemon.driver, container.ID,
container.basefs, dir)

    }

    return nil

}
```

12.4.3　initializeNetworking

函数intializeNetworking的作用与实现在7.4.2节中分析过。总而言之，Docker Daemon需要处理用户指定的网络模式(　bridge、host、other container和none模式中的一种)　，进行相应的网络配置与初始化，最终将所有初始化后的信息都存入container对象中。

12.4.4 verifyDaemonSetting

网络方面的配置，以及诸如容器内存、CPU限制等参数，都准备完毕之后，Docker Daemon对container对象的Config属性进行了进一步的验证，查看是否参数值与Docker Daemon的运行环境出现不一致的情况。函数verifyDaemonSetting就负责这样的工作，源码实现位于./docker/docker/daemon/container.go#L936-L948，如下：

```
func (container *Container) verifyDaemonSettings() {

     if container.Config.Memory > 0 && !container.daemon.sysInfo.MemoryLimit {

               log.Infof("WARNING: Your kernel does not support memory limit
capabilities. Limitation discarded.")

          container.Config.Memory = 0

     }

     if container.Config.Memory > 0 && !container.daemon.sysInfo.SwapLimit {

               log.Infof("WARNING: Your kernel does not support swap limit
capabilities. Limitation discarded.")

          container.Config.MemorySwap = -1

     }
```

```
        if container.daemon.sysInfo.IPv4ForwardingDisabled {

            log.Infof("WARNING: IPv4 forwarding is disabled. Networking will not
work")

        }

}
```

从代码的角度可见，验证的维度主要有三个：系统内核是否支持cgroup内存限制，系统内核是否支持cgroup的swap内存限制，以及系统内核是否支持网络接口间IPv4数据包的转发。首先需要明确的是：如何判断系统内核在这三个维度上是否支持。对于cgroup内存限制和swap内存限制，Docker Daemon在启动时即会前往cgroup文件系统的挂载点扫描，一旦发现memory.limit_in_bytes文件和memory.soft_limit_in_bytes文件，则说明系统内核支持cgroup内存限制；同理，一旦发现memory.memsw.limit_in_bytes，则说明系统内核支持cgroup的swap内存限制。原则上，Docker Daemon只要找到文件/proc/sys/net/ipv4/ip_forward，就可以认为系统内核支持IPv4包的转发功能，然而在Docker 1.2.0版本上却并未很好地完善这一点。

12.4.5 prepareVolumesForContainer

volume是Docker中有关存储的重要概念。了解完Docker的镜像原理之后，也许很多人都会认为Docker容器中所有内容的存储都会在aufs联合挂载的文件系统中。当然，这样的观点是不够准确的。Docker允许用户从容器外部挂载目录至容器内部，这一方面扩展了容器文件系统的范畴，另一方面拓宽了容器与外界共享资源的方式。对于单个容器而言，用户可以为其配置两种类型的volume，第一种是bind-mount类型的volume，即用户指定容器外部目录挂载到容器内部的指定目录；另一种是data volume，即用户只指定volume在容器内部的目录，关于volume在宿主机上的目录，Docker Daemon接管创建工作。

volume的存在使得Docker容器的存储可以实现持久化。对于数据存储类的镜像，如MySQL、MongDB等，均会采用data volume，用于持久化数据存储。以MySQL为例，MySQL存储引擎会将数据存储在文件系统的/var/lib/docker目录下，而此目录又是通过data volume的形式挂载进容器的，实际位置在宿主机文件系统中位于/var/lib/docker/vfs/dir/<some_id>，因此数据将会存入宿主机文件系统的/var/lib/docker/vfs/dir/<some_id>中。更为重要的是，Docker Daemon使用docker rm命令删除容器时，如果不指定-

v参数，则默认不会删除容器的data volume，即目录/var/lib/docker/vfs/dir/<some_id>。长此以往，如果不对data volume的磁盘空间进行清理，很容易消耗磁盘空间。

函数prepareVolumesForContainer的作用为：通过用户指定的volume参数以及镜像中的data volume参数，为容器准备更为具体的Volumes信息。在Docker Daemon中，结构体Volume的定义如下：

```
type Volume struct {
     HostPath    string
     VolPath     string
     Mode        string
     isBindMount bool
}
```

一个Volume实例均会有4个基本属性，HostPath代表volume在宿主机上的路径，VolPath代表volume在容器内部的路径，Mode代表容器对于Volume的读写权限，isBindMount代表volume是否为bind-mount类型。

若容器的某个volume属于bind-mount类型，则用户必须显式指定宿主机上的路径HostPath、容器内部的路径VolPath，以及想要的权限。如果volume为data volume，则data volume的规范中不允许用户指定volume在宿主机上的路径，此时Docker Daemon会接管volume在宿主机上的路径配置。Docker Daemon创建data

volume的函数为createVolumeHostPath，定义位于./docker/docker/daemon/volumes.go#L222-L238，如下：

```
func createVolumeHostPath(container *Container) (string, error) {
     volumesDriver := container.daemon.volumes.Driver()
     // Do not pass a container as the parameter for the volume creation.
     // The graph driver using the container's information ( Image ) to
     // create the parent.
     c, err := container.daemon.volumes.Create(nil, "", "", "", "", nil, nil)
     if err != nil {
          return "", err
     }
     hostPath, err := volumesDriver.Get(c.ID, "")
     if err != nil {
               return hostPath, fmt.Errorf("Driver %s failed to get volume rootfs
%s: %s", volumesDriver, c.ID, err)
     }
     return hostPath, nil
}
```

Docker Daemon通过类型为vfs的graphdriver来管理data volume，由于container.daemon.volumes是指向graph.Graph的指针，故Create函数位于graph包中，功能为创建一个镜像，并在volumes这个graph中注册，最终通过代码volumesDriver.Get（c.ID，""）返回volume在宿主机上的路径，路径为/var/lib/docker/vfs/dir/＜some_id＞/。

函数prepareVolumesForContainer完成的是mount信息的准备工作，创建所有的Volume对象，并存入container对象中。

12.4.6 setupLinkedContainers

容器间的link操作允许容器通过环境变量的形式发现另一个容器，并在这两个容器间安全传输信息。用户在启动容器时可以通过设置--link参数来完成这项工作，如docker run--link db：aliasubuntu：14.04，作用是启动容器时为容器db设置一个别名alias，并将别名alias以及容器db的网络信息配置到新启动容器的环境变量中。

函数setupLinkedContainers的作用就是为启动容器配置link的环境变量。该函数的的执行流程图如图12-4所示。

下面对于图12-4中各个步骤的作用进行简单描述。

步骤1，获取指定容器的子容器，实现代码为children，err：=daemon.Children（container.Name）。Docker创建的容器被daemon管理时并非与其他容器毫无关系。一旦容器存在link信息，Docker Daemon对于容器的“父子关系”就会充分体现出来。被链接的容器将会被视为发起链接的容器的子容器。而这样的关系将会记录在Docker Daemon管理的graphdb中。关于Docker Daemon如何并且何时记录这样的关系，实际位于container.Start（）的RegisterLinks中，有兴趣的读者可以继续深入研读Docker关于

graphdb的容器link信息记录。因此，步骤1会从graphdb中获取所需启动容器的所有子容器，目 的是为了将所有子容器的网络信息配置进此容器的环境变量。

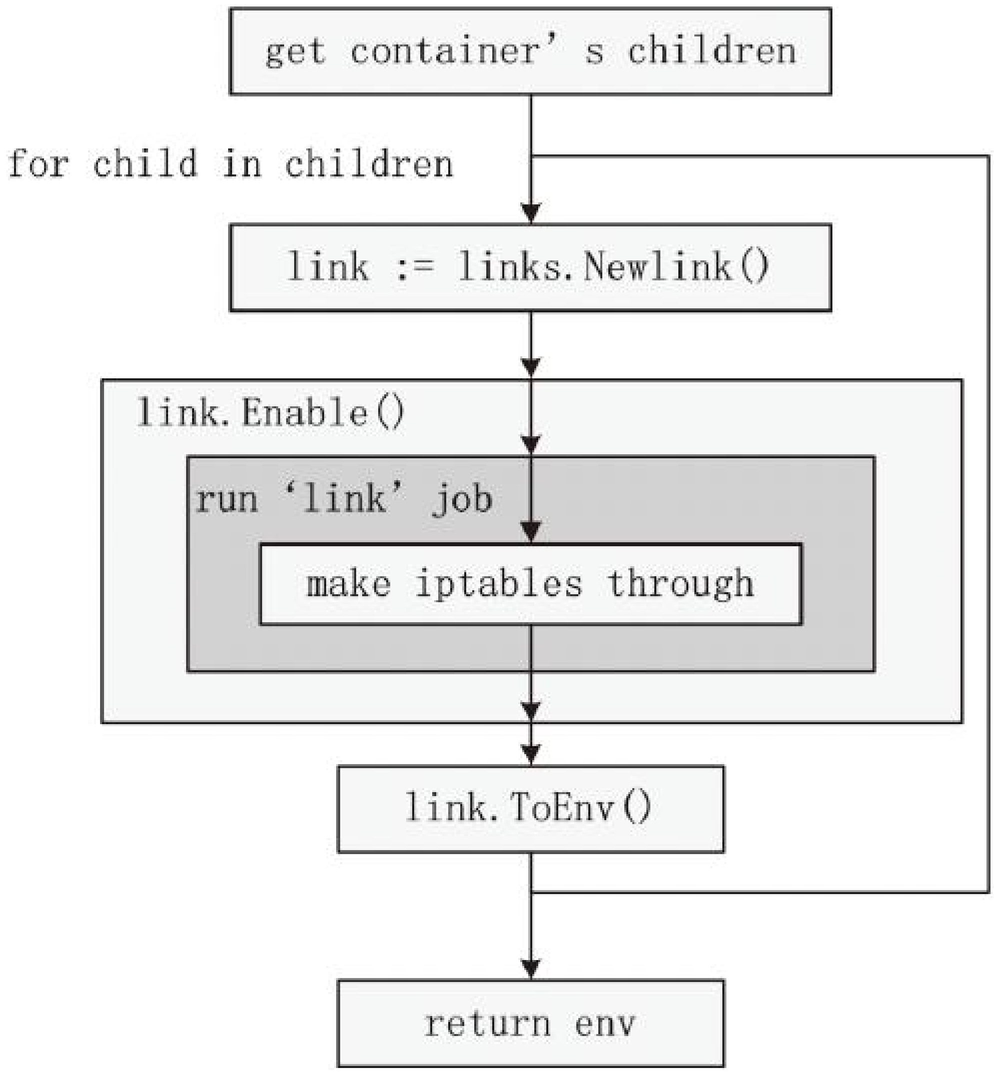


图12-4　setupLinkedContainers函数执行流程图

完成步骤1之后，Docker Daemon随即通过children中的每一个child进行link信息Env化。本节暂且分析流程中的第一个循环。

步骤2，创建完备的link对象，对象中包含link所需要的所有信息，如：父容器的IP、子容器的IP、子容器被链接时的别名、子容器Config信息中的Env属性以及子容器内部暴露的端口号。创建link对象的实现源码位于./docker/docker/daemon/container.go#L977-L983，如下：

```
link, err := links.NewLink(
     container.NetworkSettings.IPAddress,
child.NetworkSettings.IPAddress,
linkAlias,
child.Config.Env,
child.Config.ExposedPorts,
daemon.eng)
```

步骤3，使得link对象的实际意义生效，开启link容器间访问的iptables规则。link.Enable（ ） 的实现位于./docker/docker/links/link.go，实现手段为创建并执行名为“link”的job，传入的参数大多为link对象的内容。Docker Daemon启动时，执行网络初始化时，曾注册了key为“link”的处理方法，其中value为“LinkContainers”。函数LinkContainer的内容较为易懂，主要是通过链接两个容器的网络信息，保证即使在-icc=false的情况，依旧可以通过开启iptables规则，保证容器间指定端口的访问保持畅通。这样的iptables规则主要有以下两条：

```
iptables.Raw(action, "FORWARD",
     "-i", bridgeIface, "-o", bridgeIface,
     "-p", proto,
     "-s", parentIP,
     "--dport", port,
     "-d", childIP,
     "-j", "ACCEPT")
```

第一条规则用于接受父容器到子容器指定端口的连接，其中也指定了协议类型，并且表明是容器间的通信，理由是从bridgeIface（docker0）发出，从bridgeIface（docker0）接收。

```
iptables.Raw(action, "FORWARD",
     "-i", bridgeIface, "-o", bridgeIface,
     "-p", proto,
     "-s", childIP,
     "--sport", port,
     "-d", parentIP,
     "-j", "ACCEPT")
```

第二条规则用于接受子容器的指定端口到父容器的连接，其中也指定了协议类型，并且表明是容器间的通信，原因与前面一致。

步骤4，将link对象Env化，只有最终的Env信息，才能被容器内部的进程使用。实现源码为link.ToEnv（），ToEnv函数的定义位于./docker/docker/links/links.go#L50，而link对象Env化的主要源码如下：

```
if p := l.getDefaultPort(); p != nil {
          env = append(env, fmt.Sprintf("%s_PORT=%s://%s:%s", alias, p.Proto(),
           l.ChildIP, p.Port()))
     }
     // Load exposed ports into the environment
     for _, p := range l.Ports {
               env = append(env, fmt.Sprintf("%s_PORT_%s_%s=%s://%s:%s", alias,
                       p.Port(), strings.ToUpper(p.Proto()), p.Proto(),
l.ChildIP, p.Port()))
               env = append(env, fmt.Sprintf("%s_PORT_%s_%s_ADDR=%s", alias,
p.Port(), strings.ToUpper(p.Proto()), l.ChildIP))
               env = append(env, fmt.Sprintf("%s_PORT_%s_%s_PORT=%s", alias,
                  p.Port(), strings.ToUpper(p.Proto()), p.Port()))
               env = append(env, fmt.Sprintf("%s_PORT_%s_%s_PROTO=%s", alias,
                  p.Port(), strings.ToUpper(p.Proto()), p.Proto()))
     }
     // Load the linked container's name into the environment
     env = append(env, fmt.Sprintf("%s_NAME=%s", alias, l.Name))
```

以上源码也就是在父容器中出现诸多以下环境变量的原因。父容器中关于子容器link信息的样例环境变量如下：

```
DB_NAME=/web/db
DB_PORT=tcp://172.17.0.33:3456
DB_PORT_3456_TCP=tcp://172.17.0.33:3456
DB_PORT_3456_TCP_PROTO=tcp
DB_PORT_3456_TCP_PORT=3456
DB_PORT_3456_TCP_ADDR=172.17.0.33
```

众多如以上形式的env创建之后，Docker Daemon会完成所有link环境变量的拼接，在setupLinkedContainers函数返回内容时返回env，最终赋值给Start函数中的linkedEnv。

12.4.7 setupWorkingDirectory

Docker容器运行时，当前工作目录并非根目录，用户完全可以通过传入--workdir参数来设置容器运行时的当前工作目录，这部分工作由函数setupWorkingDirectory来完成。对于用户指定的workdir，Docker Daemon首先查找此路径是否存在，若存在且是目录，则表明workdir有效；否则，Docker Daemon现场创建相应的目录，并对目录添加文件权限。

12.4.8 createDaemonEnvironment

Docker容器的运行环境有很多部分组成，有通过命名空间实现的隔离环境，有通过控制组实现的资源空间环境，也有用户配置环境变量的运行环境等。函数createDaemonEnvironment则为Docker容器的运行创建相应的环境变量，这样的环境变量不仅包括系统为容器添加的环境变量，还包括用户为容器指定或生成的环境变量。

函数createDaemonEnvironment的源码实现位于./docker/docker/daemon/container.go#L1004-L1024，如下：

```
func (container *Container) createDaemonEnvironment(linkedEnv []string) []string
{

     // Setup environment

     env := []string{

          "PATH=" + DefaultPathEnv, "HOSTNAME=" + container.Config.Hostname, //
Note: we don't set HOME here because it'll get autoset intelligently // based on
the value of USER inside dockerinit, but only if it isn't // set already (ie,
that can be overridden by setting HOME via -e or ENV

          // in a Dockerfile).

     }
```

```
    if container.Config.Tty {

        env = append(env, "TERM=xterm") }

    env = append(env, linkedEnv...) // because the env on the container can
override certain default values // we need to replace the 'env' keys where they
match and append anything // else.

    env = utils.ReplaceOrAppendEnvValues(env, container.Config.Env) return env

}
```

从清晰的代码中，我们可以辨别出env对象为一个string型数组，数组中的每一项均为Docker容器的环境变量。代码表明，这样的环境变量有PATH、HOSTNAME、TERM以及用户指定容器link之后产生的环境变量。所有的环境变量会在运行容器进程时，加载到进程的环境变量中。

12.4.9 populateCommand

容器需要运行，自然离不开容器运行的入口，即通过运行程序达到容器运行的目的。为此，Docker Daemon以Command的形式提供容器的运行入口，并通过函数populateCommand填充Command的内容。

第7章对populateCommand的实现有详尽的介绍，其中较为重要的依然是Command的类型以及与其他数据结构的关系。

Docker Daemon启动容器的多个阶段都对container对象的Config属性进行配置和处理，而最终这些变化都会体现在Command对象实例中，如initializeNetworking完成的网络配置将修改Command对象中的网络部分，prepareVolumesForContainer完成的volume配置也会在setupMountsForContainer函数执行过程中配置进Command（volume转换为mount对象），setupWorkingDirectory则设置Command对象中的容器进程工作目录等。

容器运行的入口已经配置完毕，Docker Daemon启动容器也将一触即发。

12.4.10 setupMountsForContainer

函数prepareVolumesForContainer的使命是：将用户在volume接口配置的volume转换为Docker Daemon能识别的volume类型；而setupMountsForContainer的作用是：将Docker Daemon中所有需要从容器外挂载到容器内的目录，转换为execdriver可以识别的Mount类型。函数setupMountsForContainer的源码实现位于./docker/docker/daemon/volume.go#L51-L75，如下：

```
func setupMountsForContainer(container *Container) error {

     mounts := []execdriver.Mount{

          {container.ResolvConfPath, "/etc/resolv.conf", true, true}, }

     if container.HostnamePath != "" {

               mounts = append(mounts, execdriver.Mount{container.HostnamePath,
"/etc/hostname", true, true}) }

     if container.HostsPath != "" {

               mounts = append(mounts, execdriver.Mount{container.HostsPath,
"/etc/hosts", true, true}) }

     // Mount user specified volumes // Note, these are not private because you
may want propagation of (un)mounts from host // volumes. For instance if you use
```

```
-v /usr:/usr and the host later mounts /usr/share you // want this new mount in
the container for r, v := range container.Volumes {

               mounts = append(mounts, execdriver.Mount{v, r,
container.VolumesRW[r], false}) }

     container.command.Mounts = mounts return nil

}
```

以上代码中，mounts为execdriver.Mount类型的数组，数组的内容均表示容器内部的路径与宿主机路径的挂载映射关系，并标明读写权限与私有权限。容器内部的挂载点一般包括/etc/resolv.conf、/etc/hostname、/etc/hosts以及所有的volume等。最终，mounts对象赋予container.command的Mounts属性，为Command类型实例服务。

12.4.11 waitForStart

Docker Daemon为启动容器所做的准备工作不可谓不充分。准备工作全部完成之时，也是启动容器、运行容器之时。函数waitForStart的功能就是启动容器进程，并根据用户指定的重启策略应对容器启动失败的情况。

函数waitForStart的定义位于./docker/docker/daemon/container.go#L1070-L1082，如下：

```
func (container *Container) waitForStart() error {
        container.monitor = newContainerMonitor(container,
container.hostConfig.RestartPolicy)
        // block until we either receive an error from the initial start of the
container's
    // process or until the process is running in the container
    select {
    case <-container.monitor.startSignal:
    case err := <-utils.Go(container.monitor.Start):
        return err
    }
    return nil
}
```

Docker Daemon为container创建了一个containerMonitor类型实例，并将其赋予container对象的monitor属性。类型containerMonitor的定义位于./docker/docker/daemon/monitor.go，在启动容器时起到监控

容器主进程的角色。如果Docker用户为容器的启动配置了重启策略，则containerMonitor会确保主进程启动失败时基于重启策略而重启。然而，若容器由于某种原因在重启策略下依旧重启失败，则containerMonitor将重置以及清理container对象占据的资源，如为容器分配的网络资源、为容器创建的rootfs等。

理解containerMonitor的监控作用之后，还需要关注其启动容器的本质，源代码为container.monitor.Start。Start函数的作用很明显，即通过启动populateCommand函数创建的Command对象，完成容器的创建。原理如此，而实现却要更复杂一些，代码的执行流首先会进入execdriver（native），execdriver为了适配libcontainer的接口，需要根据container中的Command对象创建libcontainer的Config对象，最终通过libcontainer中的namespaces包来实现容器的启动。

创建容器对象，仅仅是为Docker容器组织了必需的配置信息，并未将配置信息落实至Docker容器运行环境中的具体资源与能力。因此，libcontainer在启动容器时，一方面为容器初始化具体的物理资源并提供相应的容器能力，另一方面启动容器的主进程。第13章将分析Docker容器主进程dockerinit以及其他相关内容。

12.5　总结

Docker容器的启动，意味着用户运行时的开始。对于Docker用户而言，仅仅能感受到自身应用的正常运行。然而，对于Docker Daemon而言，需要完成的工作要远比这复杂，从最初的镜像查找，到rootfs配备，还有网络、volume、容器link等配置信息的收集，以及最后启动代表容器主进程的Command命令。

Docker体系内的万千概念，最后都融汇在代表容器主进程的Command命令中，且最终体现在主进程的启动流程中。本章从Docker容器的创建入手，着重分析了Docker Daemon的启动全过程。

第13章　dockerinit启动

13.1　引言

Docker容器作为Docker Daemon管理的对象，给用户的体验是一个隔离性较好的运行环境。作为容器技术，隔离环境背后的技术支撑是什么？Docker容器可以运行用户应用进程，容器与进程的关系又如何？可以说，认识Docker是一条漫长的路，在这条路上，仍然存在不少疑惑。

为阐述Docker容器与进程的关系，首先以用户都见过的docker run命令为例。通过Docker容器完成指定应用进程的运行，相信所有的Docker用户都不陌生，然而，应用进程是否是容器内的第一个进程，一旦提及这一点，也会激起Docker用户心中的疑惑。虽然第7章已经对容器与进程的关系进行了初步的介绍与分析，但是关于容器内第一个进程究竟为何方神圣，依然没有揭开神秘的面纱。开门见山，Docker Daemon启动Docker容器时，运行的第一个进程并不是用户的应用进程，而是一个称为dockerinit的进程。

本章主要从源码的角度介绍dockerinit，并分析dockerinit启动流程的实现。本书基于Docker 1.2.0版本，libcontainer的版本为

1.2.0。本章内容主要包含以下4个方面：

1）简要介绍dockerinit；

2）分析dockerinit的执行入口；

3）分析dockerinit的运行；

4）分析libcontainer的运行。

13.2　dockerinit介绍

正如其名，可以认为dockerinit是Docker容器中的init进程。Linux操作系统中，init进程是一个非常重要的进程，它负责初始化系统，并且是所有用户进程的祖先进程。相同的概念，在Docker容器中也有很好的体现，dockerinit作为Docker容器中的第一个进程，同样扮演初始化容器的角色，同样是Docker容器内所有进程的祖先进程。

13.2.1　dockerinit初始化内容

既然dockerinit扮演初始化容器的角色，那么具体初始化容器的哪些内容必将受到关注。对于Docker容器而言，容器的资源以及容器的能力是重点，而分配的资源以及授予的能力，都应该在容器创建时初始化。以下是dockerinit初始化的部分资源与能力：

·网络资源。Docker Daemon为bridge（桥接）模式下的容器创建网络命名空间，创建初期命名空间内并无网络栈（如网络接口等），dockerinit需要为容器初始化网络命名空间，配置独立的网络栈；

·挂载资源。容器都有独立的mount namespace，初始化挂载资源包括设置容器内部的设备、挂载点以及文件系统等；

·用户设置。容器都属于一个用户，dockerinit负责为容器设置组（group）、组ID（GID）与用户ID（UID）；

·环境变量。容器中的环境变量使得容器内进程拥有更多的运行参数。

·容器Capability。容器中用户使用的内核与宿主机无差别，Linux的Capability机制则可以确保容器内进程以及文件的

Capability得到限制。

13.2.2 dockerinit与Docker Daemon

dockerinit是Docker容器中的第一个进程。Linux操作系统中，进程的诞生往往都由fork系统调用产生，dockerinit进程自然也不会例外。如此说来，dockerinit的父进程最有可能是Docker Daemon，现实情况的确如此。

父子关系是Docker Daemon与dockerinit之间最明了的关系。Docker Daemon完成进程fork之后，Docker Daemon是父进程，dockerinit是子进程；而exec子进程则完成容器的初始化工作。关于容器的初始化工作，需要清楚的是dockerinit的初始化依据来源于何处，例如，dockerinit需要初始化网络栈，在自身网络命名空间中创建独立的网络设备，那网络设备的配置信息从何而来？虽然新建了mount命名空间，但是命名空间的挂载点位于何处？进程dockerinit仍然需要足够的依据，保证初始化工作的顺利进行。

既然是父子关系，dockerinit被派生出来时，Docker Daemon即可以将众多的配置信息通过很多有效的途径传递至dockerinit。为了使得dockerinit得到配置信息，Docker Daemon将容器所有的Config配置先json化之后，存入本地文件系统中，而dockerinit在初

始化mount命令空间前，提取这部分信息。另外，Docker Daemon为dockerinit传递命令参数也是一种简单的方式。

13.3 dockerinit执行入口

第12章主要分析Docker容器的创建，然而，关于Docker Daemon启动容器，分析到创建monitor对象并监控容器的启动之后，却并未再深入。若继续深究，我们则可以发现程序的执行将从Docker Daemon转移至execdriver，最终进入libcontainer。

寻找dockerinit的诞生，必须追溯到execdriver。Docker Daemon启动容器的参数，最终需要libcontainer来加载，而execdriver负责将Docker Daemon描述容器的container对象转义，使其符合libcontainer的接口。在这样的过程中，execdriver调用libcontainer的namespace包时，将dockerinit作为参数传递至libcontainer。调用函数的定义位于libcontainer项目 的namespace包中，函数名为exec，源码实现位于./docker/libcontainer/namespaces/exec.go#L24，如下：

```
func Exec(container *libcontainer.Config, stdin io.Reader, stdout, stderr
io.Writer, console string, rootfs, dataPath string, args []string, createCommand
CreateCommand, startCallback func()) (int, error)
```

13.3.1 createCommand分析

函数exec传入的众多参数中，与dockerinit相关的是形参createCommand，而execdriver调用namespaces.exec（ ）函数时，实参如下：

```
func(container *libcontainer.Config, console, rootfs, dataPath, init string, child
*os.File, args []string) *exec.Cmd {
     c.Path = d.initPath
     c.Args = append([]string{
          DriverName,
          "-console", console,
          "-pipe", "3",
          "-root", filepath.Join(d.root, c.ID),
          "--",
     }, args...)
     // set this to nil so that when we set the clone flags anything else is reset
     c.SysProcAttr = &syscall.SysProcAttr{
          Cloneflags: uintptr(namespaces.GetNamespaceFlags(container.Namespaces)),
     }
     c.ExtraFiles = []*os.File{child}
     c.Env = container.Env
     c.Dir = c.Rootfs
     return &c.Cmd
}
```

可见，传入的实参为一个函数，此函数的作用是：使得libcontainer执行namespaces.exec（ ）时通过该函数创建一个exec.Cmd对象，以便libcontainer直接通过Start函数，启动该exec.Cmd对象。下面分析exec.Cmd类型实例c的各属性。

.c.Path。对于类型exec.Cmd而言，唯一不能为空的就是Path参数，它代表执行命令所在路径，而d.initPath的值即为Path路径。若Docker的版本为1.2.0，execdriver的类型为native，那么默认情况

下，d.initPath的值为/var/lib/docker/init/dockerinit-1.2.0。因此，Docker Daemon启动Docker容器时，均会执行该路径下的dockerinit-1.2.0。

.c.Args。执行命令的参数同样重要，.c.Args即代表dockerinit-1.2.0命令的执行参数。其中.c.Args是一个string数组，内容如表13-1所示。

表13-1　.c.Args参数的内容

参数	参数含义
DriverName	代表 execdriver 的具体类型，默认情况下值为“native”，此参数会在 dockerinit 执行 reexec.Init() 时使用到
- console	名为“console”的 flag，值为紧接其后的元素
console	console（pty slave）的路径
-pipe	名为“pipe”的 flag，代表同步管道的文件描述符 fd
3	名为“pipe”的 flag 的值，代表同步管道的 fd 值为 3
-root	名为“root”的 flag，代表容器配置文件所在的路径
filepath.Join(d.root, c.ID)	名为“root”的 flag 的值，一般为 /var/lib/docker/execdriver/native/<c.ID>，该路径下存放着文件 container.json 和 state.json，前者存储容器配置对象 Config 的 json 信息；后者存储容器的 init 进程信息、网络信息以及控制组路径等
args	用户指定的命令，包含 ENTRYPOINT 与 CMD，在 dockerinit 执行的后期阶段扮演重要角色，实现系统初始化工作转变为用户进程的运行

.c.SysProcAttr。SysProcAttr将携带exec.Cmd运行的操作系统参数，如在启动dockerinit时，代表是否需要为进程创建新namespace的参数Cloneflags。关于namespace的Cloneflags共有6个，分别是syscall.CLONE_NEWNS、syscall.CLONE_NEWUTS、syscall.CLONE_NEWIPC、syscall.CLONE_NEWUSER、syscall.CLONE_NEWPID和

syscall.CLONE_NEWNET，而默认情况下Docker 1.2.0并未完全支持用户命名空间，故没有使用syscall.CLONE_NEWUSER。

.c.ExtraFiles。ExtraFiles将携带从父进程Docker Daemon继承的文件描述符，由于不包括标准输入（文件描述符为0）、标准输出（文件描述符为1）和标准错误（文件描述符为2），故ExtraFile的值都是不小于3的文件描述符。由于在dockerinit的运行初期仍然需要与之通信，故Docker Daemon创建了一个管道，并将管道的一端交给dockerinit。

.c.Env。Env用于指定dockerinit进程运行时的环境变量。故用户运行docker run命令时通过-e设定的ENV参数或者Dockerfile中的ENV参数，都会在此时用来设定dockerinit进程的环境变量。

.c.Dir。Dir用于指定命令exec.Cmd的工作目录，具体的值为execdriver.Command的rootfs。

13.3.2 namespace.exec

有了createCommand的基础，理解libcontainer中的namespace.Exec就变得简单很多。namespace.Exec的作用无非是通过createCommand函数，创建一个容器启动命令exec.Cmd，随后启动该命令完成dockerinit进程的运行工作。容器命令启动的源码实现位于./docker/libcontainer/namespaces/exec.go#L24-L47，如下：

```
func Exec(container *libcontainer.Config, stdin io.Reader, stdout, stderr
io.Writer, console string, rootfs, dataPath string, args []string, createCommand
CreateCommand, startCallback func()) (int, error) {
	var (
		err error
	)
	// create a pipe so that we can syncronize with the namespaced process and
	// pass the veth name to the child
	syncPipe, err := syncpipe.NewSyncPipe()
	if err != nil {
		return -1, err
	}
	defer syncPipe.Close()
		command := createCommand(container, console, rootfs, dataPath,
os.Args[0], syncPipe.Child(), args)
	command.Stdin = stdin
	command.Stdout = stdout
	command.Stderr = stderr
	if err := command.Start(); err != nil {
		return -1, err
	}
	……
	if err := command.Wait(); err != nil {
		if _, ok := err.(*exec.ExitError); !ok {
			return -1, err
		}
	}
	……
}
```

以上代码为仅仅是namespace.Exec函数执行的前面一部分，代码结构较为清晰，总体分为三个部分：创建syncPipe、创建容器命令以及容器命令的启动。

顾名思义，创建syncPipe的作用是：使得Docker Daemon与dockerinit可以通过管道的形式同步资源。由于一旦dockerinit进程启动，dockerinit进程的运行就处于多个全新的namespace内，此时Docker Daemon与dockerinit处于相互隔离的namespace中，两者的通信变得不便（虽然dockerinit处于全新的网络命名空间下，但网络命名空间下网络栈仍未初始化，没有网络接口用于通信），故Docker采用更为底层的管道机制完成操作系统上分别处于不同namespace进程的通信。syncPipe主要用于Docker Daemon向dockerinit传递容器信息，如容器内网络设备的信息等。

创建容器命令即为createCommand函数的实现，前面已经涉及，并对最终返回的exec.Cmd类型实例进行详细分析。总而言之，createCommand函数为容器创建了静态的执行入口。

容器命令的启动通过command.Start（）实现。Start函数负责启动指定的命令command，但是不对该命令的运行执行Wait操作，Wait操作有专门的Wait函数来完成。由于command的Path参数为d.initPath，即/var/lib/docker/init/dockerinit-1.2.0，故

libcontainer会启动dockerinit-1.2.0命令，换言之，dockerinit的执行被触发。

分析至此，我们已经找到dockerinit的执行入口，但这并不意味着Docker Daemon会将所有执行权交给dockerinit，Docker Daemon关于容器的启动还远没有结束。启动dockerinit之后，Docker Daemon与dockerinit的执行将并发执行，两者之间通过管道的形式进行同步，同步完成之后，Docker Daemon与dockerinit各自运行，除父子关系之外，无更多逻辑关系。13.4节将深入分析dockerinit的执行以及dockerinit与Docker Daemon的协同通信。

13.4 dockerinit运行

dockerinit作为Docker容器中运行的第一个进程，本质上，它是一个使用go语言编译的可执行文件，即dockerinit-1.2.0。既然如此，回到dockerinit的实现函数，可以发现dockerinit的实现全部依靠reexec.Init（ ）函数。dockerinit的main函数位于./docker/docker/dockerinit/dockernit.go#L9-L12，如下：

```
func main() {
     // Running in init mode
     reexec.Init()
}
```

以上代码简洁得有点极致，main通过reexec.Init（ ）来实现。是否reexec.Init（ ）真的实现了对容器环境的初始化？让我们带着疑惑进入reexec.Init（ ）的分析。

13.4.1　reexec.Init（　）的分析

reexec.Init（　）在本书中并非第一次出现，早在第2章中，创建Docker Client时就涉及了reexec.Init（　），当时在docker的main函数的运行中，第一个步骤即检验reexec.Init（　）的值，第二个步骤才是解析flag参数，由于还没有execdriver等概念的基础，故没有深入。具体源码位于./docker/docker/docker.go#L26-L111，如下：

```
func main() {
     if reexec.Init() {
        return
     }
     flag.Parse()
     // FIXME: validate daemon flags here
     ……
}
```

Docker中包reexec的存在，主要是因为使用Go在派生进程时存在一些语言层的限制。功能上，reexec使得dockerinit这个二进制文件可以运行多种任务。dockerinit运行的处理方法都通过一个注册的execdriver名称来获取，并最终运行此处理方法。

首先进入reexec包中的Init（　）函数定义，源码实现位于./docker/docker/reexec/reexec.go#L23-L32，如下：

```
func Init() bool {
     initializer, exists := registeredInitializers[os.Args[0]]
     if exists {
          initializer()
          return true
```

```
    }
    return false
}
```

以上代码表明：Init（ ）将会执行注册的initializer。既然是注册的initializer，就会存在一个问题："Docker中哪一模块注册了相应的initializer？"。由于reexec包的功能大部分与exec相关，而在Docker架构中，与exec关系最紧密的自然是execdriver，execdriver负责执行容器的相应操作，而问题的答案恰巧的确是execdriver。

以默认的execdriver为例，也就是类型为native的execdriver，Docker Daemon启动时自然会初始化execdriver，而在初始化类型为native的execdriver时，会执行execdriver对reexec的注册，代码位于./docker/docker/daemon/execdriver/native/init.go#L19-L21，如下：

```
func init() {
    reexec.Register(DriverName, initializer)
}
```

函数init（ ）实现了DriveName的注册，DriveName的值为"native"，而注册时相应的处理方法为initializer，此initializer才是dockerinit真正运行的内容。反观createCommand环节，dockerinit的参数中第一个是DriveName（"native"），故

dockerinit运行时将直接执行key为“native”的注册处理方法。而在./docker/docker/docker.go的main函数中，一旦由于某种异常原因，reexec.Init（）已经有注册的值，而Docker Daemon启动的过程中仍然需要向reexec注册相应的execdriver，两者很有可能造成冲突。因此，docker.go中main函数确保当前没有注册execdriver之后，再继续往下执行。

13.4.2 dockerinit的执行流程

dockerinit的执行流程，可以说就是initializer的执行流程。在功能方面，initializer依旧完成一部分初始化工作，并在最后把控制权交给libcontainer的namespace，由后者完成剩余容器的初始化工作。

类型为“native”的initializer的源码实现位于./docker/docker/daemon/execdriver/native/init.go#L19-L21，如下：

```
func initializer() {
	runtime.LockOSThread()
	var (
		pipe    = flag.Int("pipe", 0, "sync pipe fd")
		console = flag.String("console", "", "console (pty slave) path")
		root    = flag.String("root", ".", "root path for configuration files")
	)
	flag.Parse()
	var container *libcontainer.Config
	f, err := os.Open(filepath.Join(*root, "container.json"))
	if err != nil {
		writeError(err)
	}
	if err := json.NewDecoder(f).Decode(&container); err != nil {
		f.Close()
		writeError(err)
	}
	f.Close()
	rootfs, err := os.Getwd()
	if err != nil {
		writeError(err)
	}
	syncPipe, err := syncpipe.NewSyncPipeFromFd(0, uintptr(*pipe))
	if err != nil {
		writeError(err)
	}
		if err := namespaces.Init(container, rootfs, *console, syncPipe,
flag.Args()); err != nil {
		writeError(err)
```

```
        }
        panic("Unreachable")
    }
```

分析initializer的实现，发现执行流程包含5个步骤，分别如下：

1） 定义flag参数并解析；

2） 声明libcontainer.Config实例，并通过解析container.json文件，获取实例内容。文件container.json：execdriver中run函数执行namespaces.exec函数之前，将libcontainer.Config对象实例持久化至本地目录/var/lib/docker/execdriver/native/<c.ID>/container.json；

3） 获取root的路径；

4） 通过同步管道所在的文件描述符索引值（值为3），获取对应的管道对象；

5） 通过libcontainer中namespaces包的Init函数，最终完成容器初始化工作。

13.5 libcontainer的运行

libcontainer是一套容器技术的实现方案，借助于libcontainer的调用，可以完成容器的创建与管理。dockerinit就通过libcontainer来完成容器的创建与初始化。

dockerinit通过namespaces包的Init函数来调用libcontainer的API接口并完成所有工作，但是Init完成的内容绝不仅仅只有与Linux namespace相关的内容。Init是全新namespace下运行的第一部分内容，同时会完成mount信息、容器用户以及网络等多方面的配置。

由于dockerinit需要与Docker Daemon进行信息同步，故几乎可以将这父子进程的管道同步作为一个分水岭，前半部分作为进程的配置，后半部作为初始化容器的资源。Init函数的实现位于./docker/libcontainer/namespaces/init.go#L32-L122，如下：

```
func Init(container *libcontainer.Config, uncleanRootfs, consolePath string,
syncPipe *syncpipe.SyncPipe, args []string) (err error) {
     ……
     if err := LoadContainerEnvironment(container); err != nil {
          return err
     }
     // We always read this as it is a way to sync with the parent as well
     var networkState *network.NetworkState
     if err := syncPipe.ReadFromParent(&networkState); err != nil {
          return err
     }
     ……
     if err := setupNetwork(container, networkState); err != nil {
          return fmt.Errorf("setup networking %s", err)
     }
```

```
        ......
        if err := mount.InitializeMountNamespace(rootfs,
             consolePath,
             container.RestrictSys,
             (*mount.MountConfig)(container.MountConfig)); err != nil {
             return fmt.Errorf("setup mount namespace %s", err)
        }
        ......
        if err := FinalizeNamespace(container); err != nil {
             return fmt.Errorf("finalize namespace %s", err)
        }
        // FinalizeNamespace can change user/group which clears the parent death
        // signal, so we restore it here.
        if err := RestoreParentDeathSignal(pdeathSignal); err != nil {
             return fmt.Errorf("restore parent death signal %s", err)
        }
        return system.Execv(args[0], args[0:], os.Environ())
   }
```

dockerinit通过向syncPipe读取父进程Docker Daemon传递的内容，完成两者信息的同步。Docker Daemon与dockerinit的同步流程如图13-1所示。

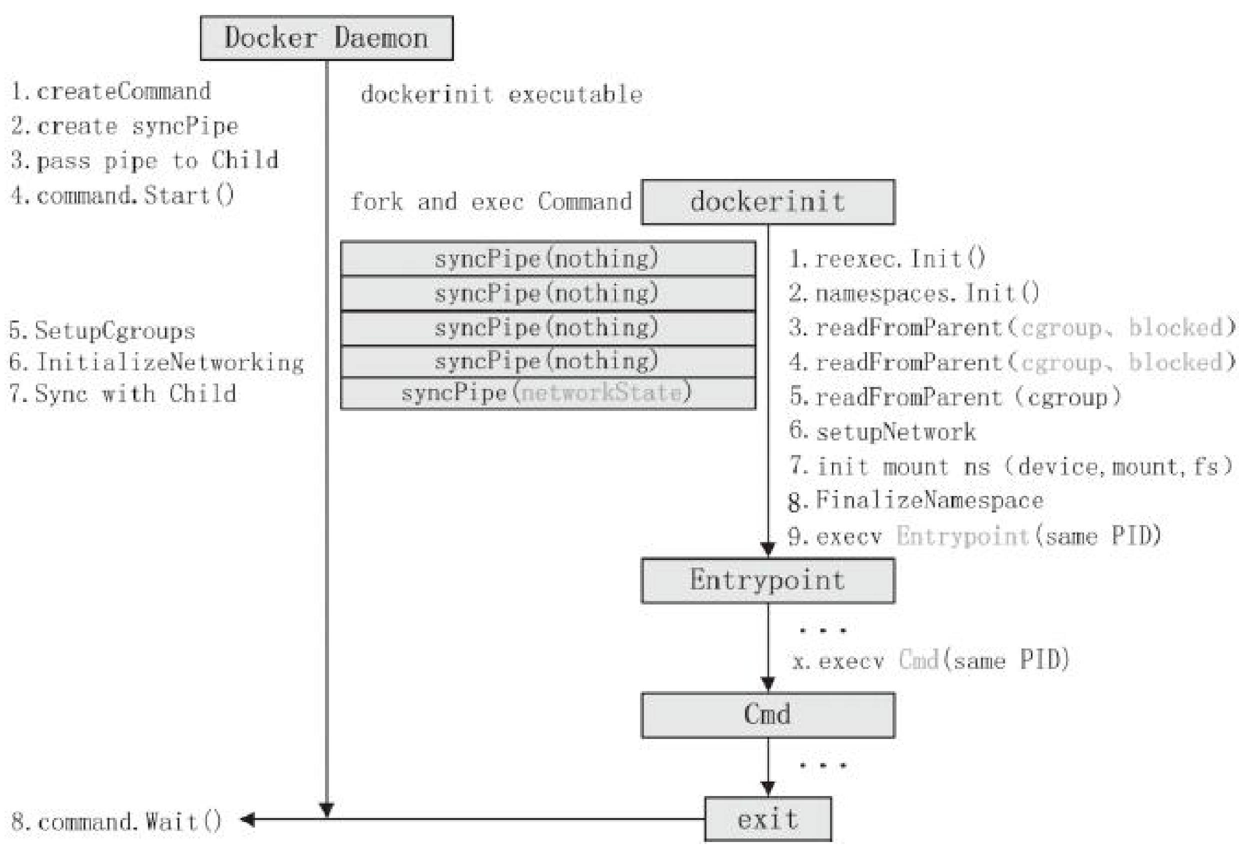


图13-1　Docker Daemon与dockerinit的同步流程

通过图13-1我们可以发现，Docker Daemon通过libcontainer中namespaces包的exec函数执行command.Start（　）之后，仍有其他的工作需要完成，主要工作如下：

1）SetupCgroups，设置dockerinit进程的cgroups，完成dockerinit进程的资源限制；

2）InitializeNetworking，创建dockerinit所在容器的网络栈资源；

3）syncPipe.ReadFromChild，与子进程dockerinit同步。

回到Docker Daemon的namespaces.Exec中，下面分析执行流程：

13.5.1　Docker Daemon设置cgroups参数

SetupCgroups通过类型为libcontainer.Config的container对象以及dockerinit进程在宿主机上的PID号，为dockerinit设置cgroups参数，从此cgroups对dockerinit进程的资源限制生效，而之后dockerinit进程派生出的子进程也将受到相同的资源限制。

资源的使用受到应有的限制，是容器技术的一大亮点。实现原理则是在进程运行之后，通过将进程的PID放置于特定的cgroups子系统来完成。

13.5.2　Docker Daemon创建网络栈资源

创建容器网络栈资源依靠函数InitializeNetworking，调用者Docker Daemon一般位于init命名空间中，而dockerinit位于新建的网络命名空间中，因此InitializeNetworking的作用是在init命名空间中创建dockerinit所在容器需要的网络栈资源，即跨越网络命名空间创建相应的网络栈。原则上，在go语言的运行时中，并不具备跨namespace的资源操纵能力，故Docker Daemon将创建完毕的网络栈资源通过syncPipe的形式传递给处于新建namespace下的dockerinit，由dockerinit负责接收并初始化新建namespace下的网络栈。关于InitializeNetworking函数的具体实现，可参考第7章。

Docker Daemon通过管道将网络栈信息传递至dockerinit之后，容器启动使命就完成了。Docker Daemon退居幕后之时，dockerinit才刚开始大展手脚。

13.5.3 dockerinit配置网络栈

虽然Docker Daemon已经完成自己的使命，但是dockerinit依旧处于阻塞状态，而且等待着SyncPipe中Docker Daemon传来的内容。只要从syncPipe中读到内容，dockerinit即从阻塞状态恢复执行。在设置SID（Session ID）、处理tty等一系列操作之后，dockerinit最为重要的工作就是配置网络栈，即通过Docker Daemon在syncPipe中传递来的NetworkState信息，配置容器所在网络命名空间下的网络栈。

首先来看NetworkState的定义，位于./docker/libcontainer/network/types.go#L34-L41，如下：

```
type NetworkState struct {
	// The name of the veth interface on the Host.
	VethHost string `json:"veth_host,omitempty"`
	// The name of the veth interface created inside the container for the child.
	VethChild string `json:"veth_child,omitempty"`
	// Net namespace path.
	NsPath string `json:"ns_path,omitempty"`
}
```

源码注释对于NetworkState各属性的描述非常到位，结合Docker的网络模式来分析，便可发现：VethHost和VethChild属于bridge网络模式下的veth pair，前者是宿主机上依附在网桥docker0上的veth，而后者是容器内部的veth；NsPath属于other container

网络模式下其他容器的网络命名空间路径。之所以NetworkState中没有host网络模式以及none网络模式的信息，原因很简单，host网络模式不需要为容器创建新的网络命名空间，而none网络模式不做任何网络配置。

以bridge网络模式为例，SetupNetwork函数正是利用NetworkState中的网络接口（ veth pair）配置信息，在容器的网络命名空间下初始化内部网络接口，将其改名为eth0，并对网络接口的MTU、IP地址以及默认网关地址进行配置。

SetupNetwork函数的实现过程中，主要通过NetworkStrategy从container.Networks对象中的Type获取所需初始化的网络接口类型，并通过具体类型进行相应的配置。若网络接口类型为veth，则网络配置工作的源码实现位于./docker/libcontainer/network/veth.go#L52-L77，如下：

```
func (v *Veth) Initialize(config *Network, networkState *NetworkState) error {
     var vethChild = networkState.VethChild
     if vethChild == "" {
          return fmt.Errorf("vethChild is not specified")
     }
     if err := InterfaceDown(vethChild); err != nil {
          return fmt.Errorf("interface down %s %s", vethChild, err)
     }
     if err := ChangeInterfaceName(vethChild, defaultDevice); err != nil {
          return fmt.Errorf("change %s to %s %s", vethChild, defaultDevice, err)
     }
     if err := SetInterfaceIp(defaultDevice, config.Address); err != nil {
          return fmt.Errorf("set %s ip %s", defaultDevice, err)
     }
     if err := SetMtu(defaultDevice, config.Mtu); err != nil {
          return fmt.Errorf("set %s mtu to %d %s", defaultDevice, config.Mtu, err)
     }
     if err := InterfaceUp(defaultDevice); err != nil {
          return fmt.Errorf("%s up %s", defaultDevice, err)
```

```
    }
    if config.Gateway != "" {
        if err := SetDefaultGateway(config.Gateway, defaultDevice); err != nil {
                return fmt.Errorf("set gateway to %s on device %s failed with
                    %s", config.Gateway, defaultDevice, err)
        }
    }
    return nil
}
```

分析以上Initialize函数的实现，很快可以总结出以下执行流程：

1）将名为VethChild的网络接口停止运行；

2）将名为VethChild的网络接口改名为defaultDevice，即eth0；

3）为网络接口eth0设置IP地址；

4）为网络接口eth0设置MTU值；

5）启动网络接口eth0；

6）若需设置网关地址，则为网络接口设置默认网关地址。

通过以上6个步骤的初始化与配置，容器网络命名空间内的网络接口即成功运行，为容器内部提供完整的网络栈，并且该veth网络接口对其他容器不可见，处于独立的容器环境中。

另外，需要说明的是，对于容器技术而言，只要新建一个网络命名空间，容器内部就会默认创建一个loopback网络环回接口，供容器

内部本地通信。

13.5.4 dockerinit初始化mount namespace

对于dockerinit而言，不仅网络命名空间下的网络栈资源需要初始化与配置，其他命名空间下的资源同样需要初始化。函数InitializeMountNamespace则用于完成挂载资源的初始化。

Docker体系中，关于挂载资源，无外乎有三种基本的资源：设备（device）、文件系统以及挂载点（mount point）。文件系统资源比较好理解，Docker容器的运行离不开文件系统，这样的文件系统自然包括容器最初的rootfs文件系统，当然也包含proc等虚拟文件系统。除此之外，如果对Docker中volume概念熟悉，理解挂载点（mount point）自然也不是难事，挂载点保证宿主机上的文件资源同样可以被挂载到容器内部，并被容器内部进程使用。设备文件的初始化，实则是为容器的设备创建一个索引节点并进行关联。

函数InitializeMountNamespace的定义位于./docker/libcontainer/mount/init.go#L29，如下：

```
func InitializeMountNamespace(rootfs, console string, sysReadonly bool,
mountConfig *MountConfig) error
```

其中mountConfig参数同样来自与container.MountConfig，而container的类型为libcontainer.Config，它正是execdriver为了

适配libcontainer通过将execdriver.Command转义后的libcontainer.Config。因此，用户态所有的mount信息，最终都会在libcontainer执行InitializeMountNamespace时有所体现。

13.5.5 dockerinit完成namespace配置

dockerinit的历史使命并非仅仅是初始化与配置容器的namespace，完成namespace的相应工作之后，仍然需要执行用户指定的Entrypoint以及Cmd。两项任务的分隔线就以函数FinalizeNamespace为准。

顾名思义，在开始执行用户命令之前，函数FinalizeNamespace完成容器namespace下所需的所有工作。函数定义位于./docker/libcontainer/namespaces/init.go#L211-L249，如下：

```
func FinalizeNamespace(container *libcontainer.Config) error {
     // Ensure that all non-standard fds we may have accidentally
     // inherited are marked close-on-exec so they stay out of the
     // container
     if err := utils.CloseExecFrom(3); err != nil {
          return fmt.Errorf("close open file descriptors %s", err)
     }
     // drop capabilities in bounding set before changing user
     if err := capabilities.DropBoundingSet(container.Capabilities); err != nil {
          return fmt.Errorf("drop bounding set %s", err)
     }
     // preserve existing capabilities while we change users
     if err := system.SetKeepCaps(); err != nil {
          return fmt.Errorf("set keep caps %s", err)
     }
     if err := SetupUser(container.User); err != nil {
          return fmt.Errorf("setup user %s", err)
     }
     if err := system.ClearKeepCaps(); err != nil {
          return fmt.Errorf("clear keep caps %s", err)
     }
     // drop all other capabilities
     if err := capabilities.DropCapabilities(container.Capabilities); err != nil
{
          return fmt.Errorf("drop capabilities %s", err)
     }
```

```
        if container.WorkingDir != "" {
            if err := syscall.Chdir(container.WorkingDir); err != nil {
                return fmt.Errorf("chdir to %s %s", container.WorkingDir, err)
            }
        }
        return nil
    }
```

以上代码逻辑全部顺序执行，逐步分析执行步骤，可以得到以下结果：

1） CloseExecFrom（3），关闭除标准输入（文件描述符为0）、标准输出（文件描述符为1）、标准错误（文件描述符为2）之外所有打开的文件描述符（文件描述符大于等于3）；

2） DropBoundingSet，在容器切换用户之前，为容器取消某些Linux Capability；

3） SetKeepCaps，在容器切换用户前，为容器保留已经拥有的Linux Capability；

4） SetupUser，为容器创建新的用户ID、组ID、补充组的ID以及用户的Home目录；

5） ClearKeepCaps，清除所有保留的Linux Capability；

6） DropCapabilities，禁用其他所有的Linux Capability；

7）syscall.Chdir（container.WorkingDir），为容器进程切换至工作目录workdir，即在用户态docker run命令指定的参数workdir。

通过以上7个步骤，dockerinit的namespace初始化工作圆满完成，一个备受瞩目的容器才真正诞生，因为容器的隔离、资源控制、权限管理等在这一刻才全部完成。dockerinit在容器的诞生阶段扮演一个先行官的角色，然而，对于最终的用户而言，dockerinit显然不是主角，用户希望自己指定的应用程序最终能在容器内按预期正常运行。

13.5.6 dockerinit执行用户命令Entrypoint

回到namespaces包的Init函数，在FinalizeNamespace之后，dockerinit立即将容器主进程的执行权交给其他程序。而这些程序的启动命令是用户在docker run命令或者Dockerfile指定的Entrypoint命令与Cmd命令。

这部分代码位于Init函数的最后部分，如下：

```
return system.Execv(args[0], args[0:], os.Environ())
```

代码言简意赅，通过Linux进程的角度来分析再适合不过。首先，从args参数中提取出第一个参数作为执行命令，第二个开始所有的参数作为第一个执行命令的运行参数，而os.Environ（ ）代表整个进程运行时的环境变量。同时，最为重要的一点是，通过系统调用执行exec操作，只是在原有进程的基础上重新执行一段程序，而并不会改变原有进程的PID，也不会创建一个新进程。system.Execv函数的定义位于./docker/libcontainer/system/linux.go#L11-L18，如下：

```
func Execv(cmd string, args []string, env []string) error {
     name, err := exec.LookPath(cmd)
     if err != nil {
          return err
     }
```

```
        return syscall.Exec(name, args, env)
}
```

在Init函数中，args参数也是在execdriver在传递createCommand函数时被同时传递至libcontainer准备。Docker体系中与args参数相关程度最高的当属Entrypoint与Cmd，一旦用户指定了Entrypoint或者Dockerfile中指定了Entrypoint，Docker Daemon在处理用户命令参数时，就会将Entrypoint的内容放在Cmd前面，一起作为最终的args；若均不存在Entrypoint，那么Docker Daemon只将Cmd作为最终的args。因此，对于一个Docker容器的配置而言，允许不存在Entrypoint，但是必须要有Cmd。在两者同时存在的情况下，启动容器时，Entrypoint更偏向于容器应用层的初始化工作，在Entrypoint中的最后一个步骤，同样会使用exec系统调用，将主进程的执行权交给Cmd，毋庸置疑，Cmd属于用户指定运行的应用程序。

虽然容器的启动经历了dockerinit、Entrypoint以及Cmd，但是这三者在运行时都基于同一个PID，而整个容器的状态全部依赖于这个PID进程的存活。另外，当Cmd在运行过程中创建其他子进程时，所有子进程都会受到相同namespace的作用，同时也会受到控制组等其他内核特性的作用。Cmd进程及其所有子进程共同构成一个用户眼中的“容器”。

总之，Docker Daemon、dockerinit、Entrypoint以及Cmd四者之间的关系如图13-2所示。

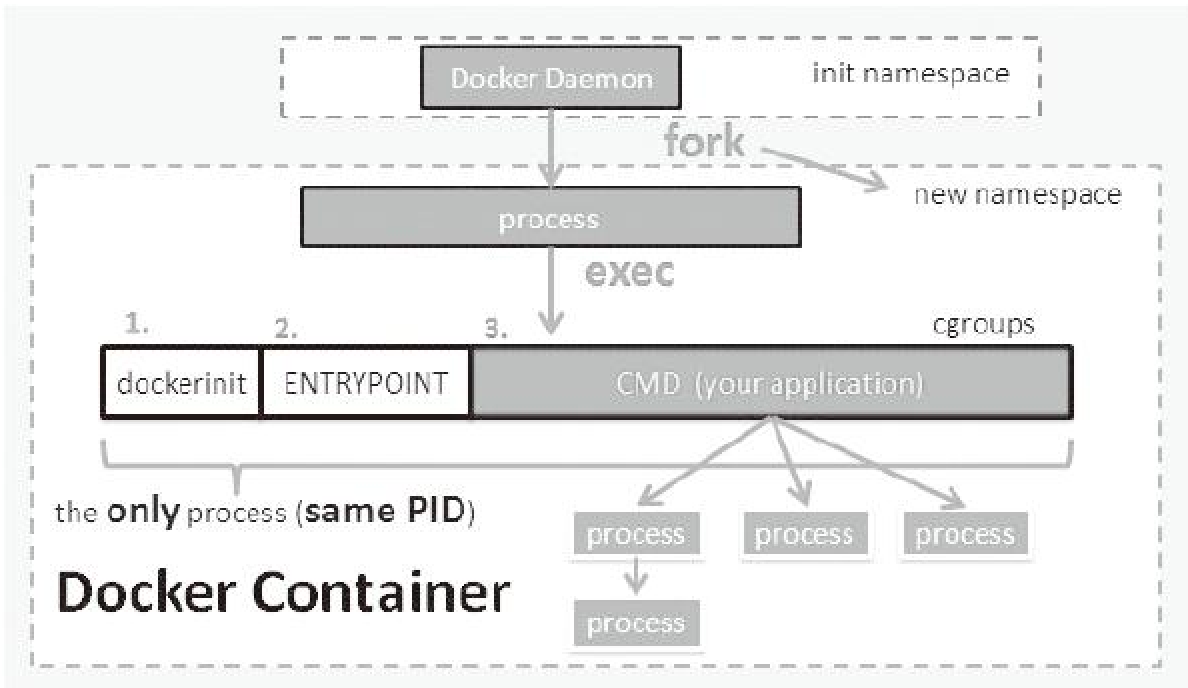


图13-2　Docker Daemon、dockerinit、Entrypoint、Cmd之间的关系

13.6　总结

深入学习dockerinit的原理，对于理解Docker容器的实现会有很大帮助。若对dockerinit思考得更多，则可以发现dockerinit的启动，即Docker容器的启动，完全是借助Linux内核的众多特性在做工作。从Linux内核的角度来看待Docker容器，容器技术的实现手段并非深不可测，容器毕竟和宿主机共享同一个Linux内核，所有工作都通过内核来完成。

Docker容器的成功创建，可以说是Docker Daemon与dockerinit协同合作的结果。Docker Daemon创建一个初步的容器环境，并将容器环境的初始化工作交给dockerinit来完成，同时两者之间完成起码的通信。dockerinit的运行起到一个承前启后的作用，它不仅接受Docker Daemon的使命，完成容器环境的初始化，同时还负责将容器的运行转变为用户应用程序Cmd的运行，并将容器交付给Docker用户。

第14章　libcontainer介绍

14.1　引言

在Docker圈，大家也许听说过这样类似于这样的话“Docker的本质并非是新颖的技术，容器技术也早已诞生多年”。可以说，此言没有半点夸大。在Docker之前，Linux平台上以LXC为代表的一系列容器技术早已在历史舞台上活跃多年。对于包括Docker在内的多种容器技术，最终的容器都离不开Linux内核的很多高级特性，如：namespace、cgroup等。

Docker架构中，Docker Daemon作为一个常驻进程，管理Docker Client请求的同时，还管理所有的Docker容器。而Linux操作系统内核态对容器的管理，自然需要为用户态的Docker Daemon服务，因此如果在Docker Daemon与Linux内核之间有一套完善的API接口，那么两者的衔接将变得尤为顺畅。Docker的发展历程中，0.9.0版本即引入了libcontainer模块，这意味着内核API接口的横空出世。

简单而言，libcontainer是一套实现容器管理的Go语言解决方案。这套解决方案实现过程中使用了Linux内核特性namespace与

cgroup，同时还采用了capability与文件权限控制等其他一系列技术。基于这些特性，除了创建容器之外，libcontainer还可以完成管理容器生命周期的任务。

Docker容器提供一个隔离、独立的环境，毫不过分地说，这样的效果完全离不开libcontainer的功劳。隔离的效果自然是容器内部与容器外部的隔离，然而，对于实现此功能的libcontainer而言，需要完成的工作正像容器内外一样，既需要在容器外做容器的配置与管理，同时还需要在容器内部完成相应的初始化工作。

另外，需要提及的是，libcontainer作为一个提供内核API接口的容器技术解决方案，设计之初就以简洁方便为目的，以至于libcontainer的生成与运行没有任何依赖。作为纯粹并且接口完备的容器管理工具，libcontainer似乎有反向定义Linux容器技术的含义，尝试成为容器技术的新生标准。同时，libcontainer的设计似乎也让开发者看到了Docker跨平台的潜在能力。Docker Daemon负责Docker Server的一系列API接口，而libcontainer接管Linux平台内核态容器技术实现的API接口，两者通过execdriver的形式协调工作。如此一来，在原有Docker Server的API接口下，似乎替换其他平台的底层容器实现技术的接口即可，如在Solaris操作系统下，开发一个类似于libcontainer的库，兼容Docker Daemon的API接口，

而实现Solaris的底层容器技术。因此，Docker今后在跨平台方面的能力绝不可低估。

14.2　Docker、libcontainer以及LXC的关系

Docker的热潮席卷全球，没有适应于如此节奏的技术爱好者，或者曾在类似于LXC技术方面有所研究的开发者，肯定会如此发问：Docker与LXC的区别到底在哪里？

从底层技术栈而言，似乎区别不大，Linux的内核特性不仅LXC可以使用，Docker可以使用，而且任何第三方的容器技术提供商均可以基于此进行开发。

从软件生命周期来讲，两者则存在很大的差异。Docker可以这样认为：一个Docker Daemon管理众多的容器，Docker Daemon为常驻的后台进程。而LXC则是一个工具，装有LXC的宿主机上并没有一个与LXC相关的常驻进程在后台运行。当用户发起与容器相关的创建与管理命令后，LXC以进程的形式来处理命令，命令完成后进程立即退出。

既然Docker最终也需要创建与管理容器，那么是否也需要类似于LXC的工具呢？答案是肯定的。事实上，Docker 0.9.0以前Docker正是通过LXC这套工具来完成底层容器的创建与管理；而从Docker 0.9.0开始Docker优先选择libcontainer作为创建以及管理容器的工

具。Docker、libcontainer以及LXC在Docker架构中的关系如图14-1所示。

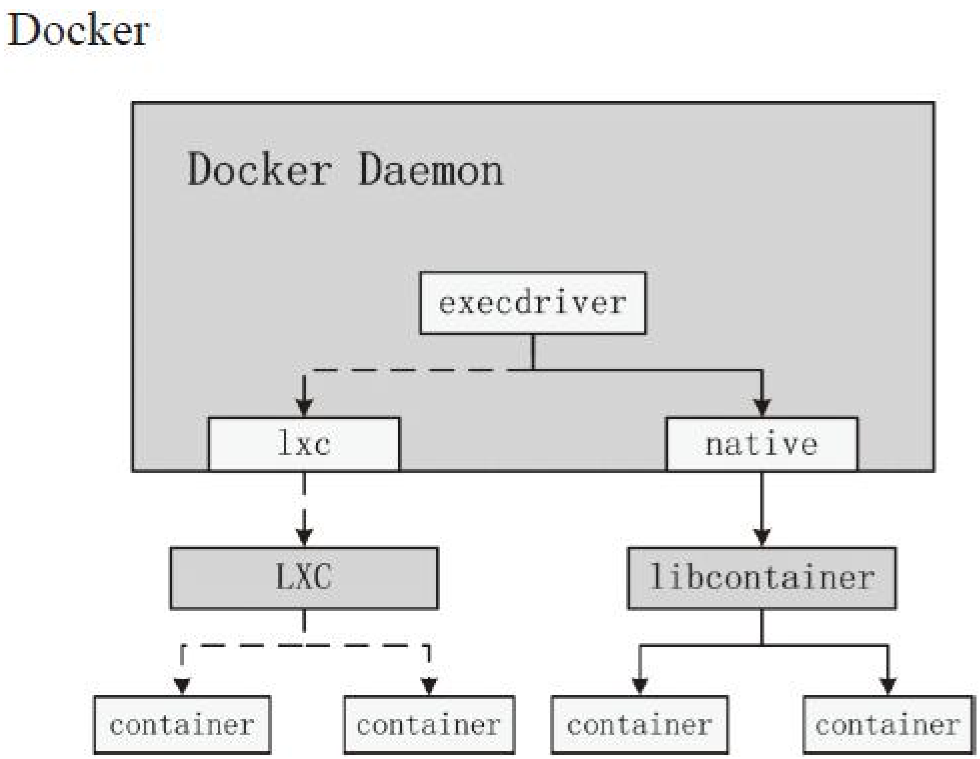


图14-1　Docker、LXC与libcontainer的关系

14.3 libcontainer模块分析

Docker容器范畴内一切与Linux内核相关的技术实现，都可以认为是通过libcontainer来完成的。libcontainer指定了所有与容器相关的配置信息，首先自然是namespace与cgroup信息，除此之外，还有容器的网络栈配置信息，当然，还包括容器的挂载点配置、设备信息等。为实现与Linux内核的通信，libcontainer还包含模块netlink，完成内核态进程与用户态进程的IPC（进程间通信）等。

本节主要介绍libcontainer的模块组成与各模块功能，对libcontainer的分析均基于libcontainer 1.2.0版本。

14.3.1　namespace

namespace是libcontainer的左膀右臂，可以说没有隔离即不存在容器。对于一个容器而言，从它诞生的那一刻起，namespace就已经开始为容器发挥作用了。Linux操作系统中，创建进程一般会使用fork等系统调用。创建进程时，fork等系统调用完成子进程对父进程task_struct的复制，且fork系统调用时可以传入众多与namespace相关的flag参数。而Linux操作系统通过exec来启动子进程的运行。回到容器的创建，由于容器的本质也是进程，故在派生进程时，Docker Daemon传入了与namespace相关的flag参数，实现namespace的创建，确保容器(　进程)　被执行之后，也就是在进程在运行时达到namespace隔离的效果。

目前在namespace方面，libcontainer有以下支持，源代码位于./docker/libcontainer/namespaces/types_linux.go#L7-L16，如下：

```
func init() {
	namespaceList = Namespaces{
		{Key: "NEWNS", Value: syscall.CLONE_NEWNS, File: "mnt"},
		{Key: "NEWUTS", Value: syscall.CLONE_NEWUTS, File: "uts"},
		{Key: "NEWIPC", Value: syscall.CLONE_NEWIPC, File: "ipc"},
		{Key: "NEWUSER", Value: syscall.CLONE_NEWUSER, File: "user"},
		{Key: "NEWPID", Value: syscall.CLONE_NEWPID, File: "pid"},
		{Key: "NEWNET", Value: syscall.CLONE_NEWNET, File: "net"},
	}
}
```

值得一提的是，虽然namespace中加入了用户命名空间，但是libcontainer由于一些特殊原因却并未将其完全实现。另外，对于网络命名空间的支持，也要视用户对容器的需求而定，若用户指定容器的网络模式为host模式，则libcontainer不为容器创建新的网络命名空间。

namespace的完全生效，并不仅仅依靠namespace的创建，同时还需要对每一个namespace进行初始化。如mount命名空间、网络命名空间、uts命名空间等命名空间需要Docker Daemon提供初始化信息，而其他命令空间则采用默认值。

1.namespace的创建

为容器创建namespace完全是Docker Daemon的职责范畴，Docker Daemon一方面从自身所在namespace创建新的namespace服务于容器，另一方面在容器namespace之外，为容器配置namespace之内所需的命名空间资源。libcontainer的源码中，内部定义的exec函数实现了这部分内容，位于./docker/libcontainer/namespaces/exec.go#L24：

```
func Exec(container *libcontainer.Config, stdin io.Reader, stdout, stderr
io.Writer, console string, rootfs, dataPath string, args []string, createCommand
CreateCommand, startCallback func()) (int, error) {
     ...
     syncPipe, err := syncpipe.NewSyncPipe()
     ...
          command := createCommand(container, console, rootfs, dataPath,
os.Args[0],
            syncPipe.Child(), args)
```

```
	...
	if err := command.Start(); err != nil {
		return -1, err
	}
	...
	started, err := system.GetProcessStartTime(command.Process.Pid)
	if err != nil {
		return -1, err
	}
	...
	cgroupRef, err := SetupCgroups(container, command.Process.Pid)
	...
	var networkState network.NetworkState
		if err := InitializeNetworking(container, command.Process.Pid,
syncPipe,
		&networkState); err != nil {
		command.Process.Kill()
		command.Wait()
		return -1, err
	}
	state := &libcontainer.State{
		InitPid:       command.Process.Pid,
		InitStartTime: started,
		NetworkState:  networkState,
		CgroupPaths:   cgroupPaths,
	}
	if err := libcontainer.SaveState(dataPath, state); err != nil {
		command.Process.Kill()
		command.Wait()
		return -1, err
	}
	defer libcontainer.DeleteState(dataPath)
	// Sync with child
	if err := syncPipe.ReadFromChild(); err != nil {
		command.Process.Kill()
		command.Wait()
		return -1, err
	}
	...
	return command.ProcessState.Sys().(syscall.WaitStatus).ExitStatus(), nil
}
```

Libcontainer中namespace的exec实现，具体完成的主要工作有：

1） 创建syncpipe，以便后续Docker Daemon与容器进程跨namespace进行信息传递；

2） 创建容器内部第一个进程的可执行命令；

3）启动该命令实现namespace的创建；

4）为容器的第一个进程进行cgroup的限制；

5）在Docker Daemon所在namespace中初始化容器内部所需的网络资源，以便后续通过管道的形式将资源传递至容器内部；

6）通过管道跨namespace将网络资源传递至容器进程。

2.namespace初始化

Docker Daemon将容器所需要的资源配置信息通过两种途径传递至容器，分别为：携带网络资源信息的syncpipe以及Docker Daemon持久化到宿主机的container.json文件。第13章分析过容器内第一个进程为dockerinit，正是dockerinit完成了容器自身namespace内部挂载资源、用户资源、网络资源等的初始化工作。该部分内容实现位于./docker/libcontainer/namespaces/init.go#32-L122：

```
func Init(container *libcontainer.Config, uncleanRootfs, consolePath string,
syncPipe *syncpipe.SyncPipe, args []string) (err error) {
     ...
     if err := LoadContainerEnvironment(container); err != nil {
          return err
     }
     // We always read this as it is a way to sync with the parent as well
     var networkState *network.NetworkState
     if err := syncPipe.ReadFromParent(&networkState); err != nil {
          return err
     }
     ...
     if _, err := syscall.Setsid(); err != nil {
          return fmt.Errorf("setsid %s", err)
```

```
        }
        ...
        if err := setupNetwork(container, networkState); err != nil {
             return fmt.Errorf("setup networking %s", err)
        }
        ...
        if err := mount.InitializeMountNamespace(rootfs,
             consolePath,
             container.RestrictSys,
             (*mount.MountConfig)(container.MountConfig)); err != nil {
             return fmt.Errorf("setup mount namespace %s", err)
        }
        ...
        if err := label.SetProcessLabel(container.ProcessLabel); err != nil {
             return fmt.Errorf("set process label %s", err)
        }
        ...
        if err := FinalizeNamespace(container); err != nil {
             return fmt.Errorf("finalize namespace %s", err)
        }
        ...
        return system.Execv(args[0], args[0:], os.Environ())
   }
```

容器的整个生命周期中，namespace是最为基础的一块。Docker容器的很多功能均在namespace的基础上完成。简单的理解是：隔离的环境创建完毕之后，内部环境的初始化才能登场。

14.3.2 cgroup

cgroup是Linux内核的一个特性，此特性可以帮助用户对一组进程进行资源使用的限制、统计以及隔离。Docker领域也不例外，Docker利用对cgroup的支持，完成对Docker容器进程组的资源限制、统计以及隔离。

与cgroup内容相关的Cgroup定义，位于./docker/libcontainer/cgroups/cgroups.go#L40-L55，如下：

```
type Cgroup struct {

     Name   string

     Parent string                      // name of parent cgroup or slice
AllowAllDevices   bool          // If this is true allow access to any kind of
device within the container. If false, allow access only to devices explicitly
listed in the allowed_devices list.

     AllowedDevices    []*devices.Device

     Memory          int64     // Memory limit (in bytes) MemoryReservation int64
// Memory reservation or soft_limit (in bytes) MemorySwap      int64      //
Total memory usage (memory + swap); set `-1' to disable swap

          CpuShares         int64     // CPU shares (relative weight vs. other
containers)

          CpuQuota         int64     // CPU hardcap limit (in usecs). Allowed cpu
time in a given period.
```

```
        CpuPeriod        int64     // CPU period to be used for hardcapping
(in usecs). 0 to use system default.

     CpusetCpus        string     // CPU to use Freezer           FreezerState
// set the freeze value for the process Slice            string     // Parent
slice to use for systemd }
```

从结构体Cgroup的定义可见，Docker对于cgroup的支持，主要有以下5方面：设备(device) 、内存(Memory) 、CPU、Freezer以及systemd。在容器设备方面，Docker支持让用户选择容器可以使用的设备。在内存方面，支持为容器的运行设定用量限额。在CPU方面，支持容器进程之间拥有相对的运行时间片。Freezer则可以使容器挂起，节省CPU资源，Docker命令中pause与unpause命令即采用了cgroup的Freezer子系统。需要注意的是，容器进程组挂起，并不意味着进程已经终止，从Linux内核的角度来看，被挂起的进程拥有完整的task_struct，占用相应的内存，但是Freezer子系统保证容器中的进程不会被CPU调度到。Slice属性则属于systemd方面的配置。

14.3.3 网络

Docker的网络一直是一个备受关注的技术点。目前，关于容器网络的管理，底层实现全部借助libcontainer的network包来完成。network包定义了Docker容器的网络栈类型。Docker容器的网络模式有：bridge模式、host模式、other container模式以及none模式。这4种网络模式对网络环境的配置分别如下。

·bridge模式：为容器配置veth网络接口对，并配置loopback接口；

·host模式：不为容器创建网络命名空间；

·other container模式：不为容器创建网络命名空间，让容器与其他容器共享网络命名空间；

·none模式：为容器创建网络命名空间，配置loopback接口。

libcontainer为了标识网络接口的数据接口，抽象出network结构体，位于./docker/libcontainer/network/types.go#L7-L30，定义如下：

```
type Network struct {
     // Type sets the networks type, commonly veth and loopback
     Type string `json:"type,omitempty"`
```

```
    // Path to network namespace
    NsPath string `json:"ns_path,omitempty"`
    // The bridge to use.
    Bridge string `json:"bridge,omitempty"`
    // Prefix for the veth interfaces.
    VethPrefix string `json:"veth_prefix,omitempty"`
    // Address contains the IP and mask to set on the network interface
    Address string `json:"address,omitempty"`
    // Gateway sets the gateway address that is used as the default for the
interface
    Gateway string `json:"gateway,omitempty"`
        // Mtu sets the mtu value for the interface and will be mirrored on
both the host and
        // container's interfaces if a pair is created, specifically in the
case of type veth
    // Note: This does not apply to loopback interfaces.
    Mtu int `json:"mtu,omitempty"`
}
```

Network结构体代表容器的网络接口。容器网络接口类型（Type）一般有两种：veth和loopback；NsPath代表容器网络命名空间的所在路径；Bridge代表容器网络使用的网桥设备名；VethPrefix代表容器veth网络接口名的前缀；Address和Gateway分别代表容器网络接口的IP地址以及网关地址；最后，Mtu则代表容器网络接口设备的MTU值。

14.3.4 挂载

文件系统的使用，Docker容器绝不仅限于镜像所提供的rootfs。volume的存在（包括bind-mount以及data volume）使得容器的文件系统可以共享宿主机的资源。除了volume之外，容器不少有关于容器信息的配置文件，也是通过挂载（mount）的形式使得容器可以使用mount命名空间之外的资源，这样的配置文件包括hostname、hosts以及resolv.conf。

libcontainer对于Mount对象的定义位于./docker/libcontainer/mounts/types.go#L28-L35，如下：

```
type Mount struct {
          Type        string `json:"type,omitempty"`
          Source      string `json:"source,omitempty"`
     // Source path, in the host namespace
          Destination string `json:"destination,omitempty"`
     // Destination path, in the container
     Writable    bool   `json:"writable,omitempty"`
          Relabel     string `json:"relabel,omitempty"`
     // Relabel source if set, "z" indicates shared, "Z" indicates unshared
     Private     bool   `json:"private,omitempty"`
}
```

对于Docker Daemon而言，记录Mount的源地址以及目 标地址无疑是最重要的。在此基础上，再对Mount对象进行一些属性配置，如Mount对象是否开启容器的写权限等。

14.3.5　设备

对于容器而言，使用Linux内核管辖范围内的设备，是一个非常基本的需求。libcontainer则使用devices包来为Docker容器提供相应的设备。

libcontainer对于设备的定义位于./docker/libcontainer/devices/devices.go#L20-L27，如下：

```
type Device struct {
     Type              rune        `json:"type,omitempty"`
          Path              string      `json:"path,omitempty"`
     // It is fine if this is an empty string in the case that you are using
Wildcards
          MajorNumber       int64       `json:"major_number,omitempty"`
     // Use the wildcard constant for wildcards.
          MinorNumber       int64       `json:"minor_number,omitempty"`
     // Use the wildcard constant for wildcards.
          CgroupPermissions string      `json:"cgroup_permissions,omitempty"`
     // Typically just "rwm"
          FileMode          os.FileMode `json:"file_mode,omitempty"`
     // The permission bits of the file's mode
}
```

默认情况下，libcontainer会为容器创建某些必备的设备，如/dev/null、/dev/zero、/dev/full、/dev/tty、/dev/urandom、/dev/console等。

14.3.6 nsinit

libcontainer的存在，使得Docker操纵Linux内核特性从而管理容器成为可能。既然如此，在逆向思维下，不少人肯定会有疑问：如果仅仅只有libcontainer，而没有更多诸如Docker Daemon之类的软件，是否依然可以创建容器？

答案的是肯定的，nsinit就是最好的例子。nsinit是一个功能强大的应用程序，该应用程序巧妙利用了libcontainer，使得容器的创建变得异常简单。若需要创建一个容器，则按照libcontainer的API接口，必须要向其提供容器的rootfs以及容器的参数配置。因此，nsinit的运行需要提供一个rootfs以及一个容器的配置文件container.json。JSON文件container.json需要处于rootfs的根目录下，并且文件中含有的配置信息需要包括：容器的环境变量、网络和容器Capability等内容。

14.3.7 其他模块

libcontainer的组成较为丰富，除了以上这些较为重要的组成部分之外，其他模块的存在也极大地完善着libcontainer的功能。

libcontainer完成很多容器内部的配置工作，如何真正与Linux内核打交道，肯定是libcontainer必须要完成的工作。Netlink作为Linux内核的一套接口，提供进程间用户态与内核态之间的通信方式。

在容器的安全方面，libcontainer由security模块负责。Linux的Capability机制作为一项非常重要的安全指标，Docker可以由用户来自定义配置容器的Capability，具体实现由security包完成。

Capability的存在极大地限制了用户进程的权限，然而，安全的范畴太广，不可能全部倚仗Capability。由于Docker的namespace没有完全实现用户命名空间，因此Docker容器中的超级管理员用户root与宿主机上的root用户是同一个用户。没有隔离用户，的确是一个很大的安全隐患，然而，Docker通过Capability特性尽量降低这方面的影响。可以说，使用了Capability特性之后，在权限方面，Docker容器内的root用户和宿主机上的root用户有很大的差别，容器

内root用户的权限自然会小很多。与此同时，Docker还支持SELinux以及Apparmor，为容器的安全保驾护航。

14.4　总结

开发者普遍了解甚至掌握Docker，却往往会忽视libcontainer的重要性。libcontainer作为容器技术的一种解决方案，很好地衔接了Docker Daemon这个管理引擎与Linux的内核特性。更为直接的评价是：Docker的运行不能没有libcontainer，而在libcontainer的基础上，任何开发者或团队都可以开发或者再造类似Docker的容器管理引擎。相信Docker官方也希望更多开发者参与libcontainer的维护工作，并努力推动其发展，使libcontainer成为容器技术的标准，从而在容器市场上占据更大的份额。

第15章　Swarm架构设计与实现

15.1　引言

Docker自诞生以来，其容器特性以及镜像特性给DevOps爱好者带来了诸多方便。然而，在很长的一段时间内，Docker只能在单宿主机上运行，其跨宿主机的部署、运行与管理能力颇受外界诟病。跨宿主机能力的薄弱，直接导致Docker容器与宿主机的紧耦合，这种情况下，Docker容器的灵活性很难令人满意，容器的迁移、分组等都成为很难实现的功能点。

Swarm是Docker公司在2014年12月初新发布的容器管理工具。和Swarm一起发布的Docker管理工具还有Machine以及Compose。

Swarm是一套较为简单的工具，用于管理Docker集群，使得Docker集群暴露给用户时相当于一个虚拟的整体。Swarm使用标准的Docker API接口作为其前端访问入口，换言之，各种形式的Docker Client（Go语言的客户端、docker_py、docker等）均可以直接与Swarm通信。Swarm几乎全部用Go语言来完成开发，并且还处于一个Alpha版本。然而，Swarm的发展十分快速，功能和特性

的变更迭代还非常频繁。因此，可以说Swarm还不推荐用于生产环境中，但可以肯定的是Swarm是一项很有前途的技术。

Swarm的设计初衷和其他Docker项目 一样，遵循“batteries included but removable”原则。笔者对该原则的理解是：batteries included代表设计Swarm时，为了完全体现分布式容器集群部署、运行与管理功能的完整性，Swarm和Docker协同工作，Swarm内部包含了一个较为简易的调度模块，以达到对Docker集群调度管理的效果；“but removable”意味着Swarm与Docker并非紧耦合，同时Swarm中的调度模块同样可以定制化，用户可以按照自己的需求，将其替换为更为强大的调度模块，如Mesos等。另外，这套管理引擎并未侵入Docker的使用，这套机制也为其他容器技术的集群部署、运行与管理方式提供了思路。

本章主要从Swarm 0.2.0源码的角度分析Swarm的架构设计与实现。分析内容包含以下三个步骤：

1） Swarm架构；

2） Swarm命令；

3） Swarm运行。

15.2　Swarm架构

Swarm作为一个管理Docker集群的工具，我们必须首先将其部署于某一节点。Swarm的部署形式比较丰富，我们可以将其部署于物理服务器上，也可以将其部署于虚拟机中，还可以将其部署在Docker容器中。

由于Swarm用于管理Docker集群，因此我们自然需要一个或者多个Docker集群，集群中每一个节点均安装并运行着Docker。具体的Swarm架构如图15-1所示。

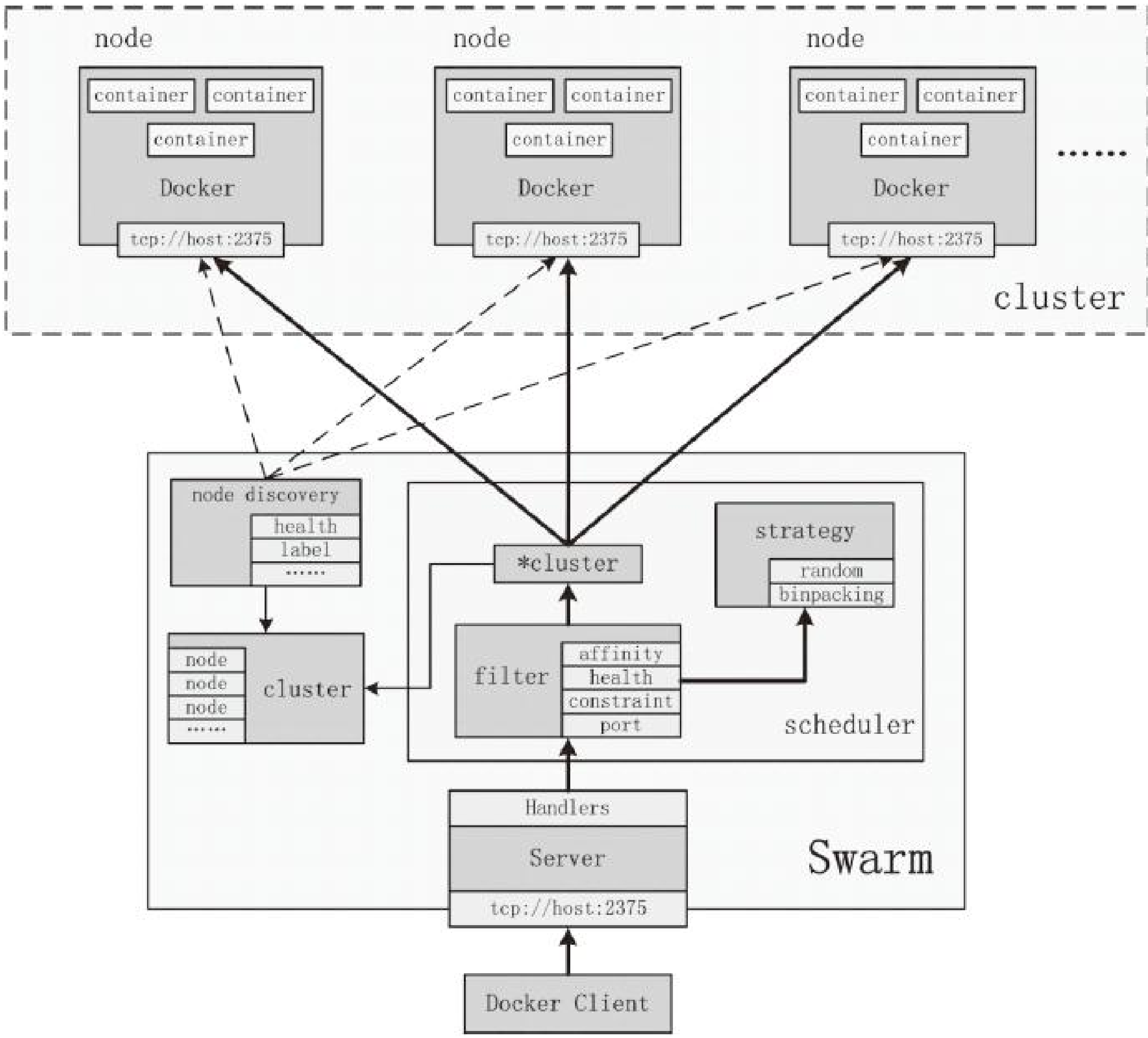


图15-1　Swarm架构

Swarm架构中最主要的计算部分是Swarm节点，Swarm管理的对象自然是Docker Cluster，Docker Cluster由多个Docker Node组成，而负责给Swarm发送请求的是Docker Client。简单而言，Swarm与Docker一样，也是C/S架构，Client为Docker Client，Server是Swarm进程。以下主要介绍Swarm Node、Docker Node、node discovery以及scheduler这4个重要的概念。

15.2.1 Swarm Node

Swarm Node可以认为是Swarm的主控节点，角色由运行的Swarm程序来充当。Swarm Node正常运行之后，该节点可以按需完成容器调度的任务，最终达到管理Docker Node集群的效果。

一般情况下，用户会通过添加标签（label）的形式，为Docker Node集群中的每一个节点设置角色。当需要调度Docker容器时，Swarm Node根据用户需求以及所有Docker Node的角色，决策出最佳的Docker Node方案，并将容器调度至该Docker Node。

为完成以上使命，Swarm的设计者将Swarm Node设计成多个模块的形式。模块各司其职，并有机结合。Swarm接收用户请求的模块暂且可以称为Server模块；用于发现集群中Docker Node的模块，我们可以将其抽象为node discovery；收集Docker Node角色，处理请求，调度容器的任务则最终落在scheduler身上。对于Swarm而言，scheduler的重要性不言而喻。更为细化地分析scheduler，可以发现scheduler内部还有负责筛选Docker Node的filter模块，以及在筛选后的集合中如何做决策的strategy模块。

15.2.2　Docker Node

Swarm管理Docker集群，Docker集群则由一个或多个Docker Node组成。Docker Node指的是运行Docker Daemon的计算节点。对于Docker Node而言，只要有足够的权限操纵Docker Daemon，整个Docker Node的容器管理能力就被掌控。Swarm Node就利用了这一点。Swarm通过tcp的方式，访问Docker Node监听的tcp端口，从而控制Docker Node。原则上，无论何时Docker管理员都应该尽量避免Docker Daemon监听tcp端口，以便带来安全隐患。特殊情况下，应该在TLS协议的保障下，开启Docker Daemon的tcp监听端口。

对Docker Node而言，与Swarm Node建立联系是其融入集群的第一步。然而，这还不够，还不能满足用户容器的按需调度。为实现有效合理的容器调度功能，Swarm建议Docker Node使用标签的形式来标记自身的Docker Daemon，使得Swarm获取Docker Daemon信息时记录Docker Node的角色。

15.2.3 node discovery

node discovery属于Swarm架构中的服务发现机制。获知Docker集群中的Docker Node数量以及每个Docker Node具体的信息，关乎Swarm的管理效率以及集群的扩展能力。

服务发现的作用很明显，只有向Swarm Node注册过的Docker Node才会被Swarm划入Docker集群范畴并管理。Swarm提供多种有效的服务注册方式，每当Docker Node注册完毕之后，Swarm有能力通过注册信息与Docker Node上的Docker Daemon进程建立通信，实现Docker Node的角色信息获取等。

15.2.4　scheduler

scheduler属于Swarm Node的调度器。对于每一个容器的调度请求，scheduler都有义务为其确定候选的Docker Node，并对这些Docker Node按照用户指定的策略进行排序。目前，Swarm 0.2.0中，scheduler的策略有三种：spread、binpack以及random。其中，spread和binpack策略会权衡每一个候选Docker Node的可用CPU资源、可用内存资源以及当前Docker Node上运行容器的数量。而random则简单得多，该策略不做任何计算，仅仅随机挑选一个候选Docker Node。

15.3　Swarm命令

通过Swarm架构图，大家可以对Swarm有一个初步的认识，比如Swarm的具体工作流程：Docker Client发送请求给Swarm；Swarm处理请求并发送至相应的Docker Node；Docker Node执行相应的操作并返回响应。除此之外，Swarm的工作原理依然还不够明了。

深入理解Swarm的工作原理，可以先从Swarm提供的命令入手。Swarm命令主要有4个：swarm create、swarm manage、swarm join、swarm list。当然，还有一个swarm help命令，该命令有助于正确使用swarm命令，本章不再赘述。

15.3.1 swarm create

Swarm中，swarm create命令用于创建一个集群标志。当Swarm管理Docker集群时，Docker Node通过这个全球唯一的集群标志实现节点注册功能，Swarm通过该标志发现集群中的Docker Node。

发起该命令之后，Swarm会前往Docker Hub上内建的服务发现中获取一个全球唯一的token，用于唯一地标识Swarm管理的Docker集群。

Swarm的运行需要使用服务发现，目前该服务内建于Docker Hub，该服务发现机制仍处于alpha版本，站点为：http://discovery-stage.hub/docker.com。

15.3.2　swarm manage

Swarm中，swarm manage是最为重要的管理命令。一旦swarm manage命令在Swarm Node上被触发，则说明用户开始使用swarm管理Docker集群。从运行流程的角度来讲，swarm经历的阶段主要有两个：启动swarm，接收并处理Docker集群管理请求。

Swarm的启动过程包含三个步骤：

1）发现Docker集群中的各个Docker Node，收集节点状态、角色信息，并监视节点状态的变化；

2）初始化内部调度器模块；

3）创建并启动API监听服务模块；

第一步，Swarm发现Docker集群中的节点。发现（discovery）是Swarm中用于维护Docker集群状态的机制。既然涉及发现，那么在这之前必须先有注册（register）。Swarm中有专门负责发现的模块，而关于注册部分，不同的discovery模式下，注册也会有不同的形式。

目前，Swarm中提供了5种不同的发现机制：Node Discovery、File Discovery、Consul Discovery、EtcD Discovery和Zookeeper Discovery。

第二步，Swarm内部的调度器模块被初始化。当swarm通过发现机制发现所有注册的Docker Node并收集到所有Docker Node的状态以及具体信息后，一旦Swarm接收到具体的Docker管理请求，Swarm就需要对请求进行处理，并通过所有Docker Node的状态以及具体信息，来筛选决策到底哪些Docker Node满足要求，并通过一定的策略将请求转发至具体的Docker Node。

第三步，Swarm创建并初始化API监听服务模块。从功能的角度来讲，可以将该模块抽象为Swarm Server。需要说明的是：虽然Swarm Server完全兼容Docker的API，但是有不少Docker命令目前是不支持的，毕竟管理Docker集群与管理单独的Docker Daemon会有一些区别。当Swarm Server初始化完毕并完成监听之后，用户即可以通过Docker Client向Swarm发送Docker集群的管理请求。

总之，Swarm的swarm manage接收并处理Docker集群的管理请求，这也是Swarm内部多个模块协同合作的结果：请求入口为Swarm Server，处理引擎为scheduler，节点信息Discovery。

15.3.3　swarm join

Swarm的swarm join命令用于将Docker Node添加至Swarm管理的Docker集群中。从功能上出发，我们不难发现swarm join命令的执行位于Docker Node，因此在Docker Node上运行该命令，首先必须安装Swarm。由于Docker Node上的Swarm只可能执行swarm join命令，故我们可以将其看成是Docker Node上用于注册的agent模块。

就功能而言，可以认为swarm join完成Docker Node在Swarm节点处的注册（register）工作，以便Swarm在执行swarm manage时可以发现该Docker Node。然而，15.3.2节提及的5种discovery机制中，并非每种机制都支持swarm join命令。不支持的discovery机制有Node Discovery与File Discovery。

Docker Node上swarm join命令的执行，标志着Docker Node向Swarm注册。Swarm通过注册信息，发现Docker Node，并获取Docker Node的状态以及具体信息，以便处理Docker请求时作为调度依据。

15.3.4 swarm list

Swarm中的swarm list命令用于列举Docker集群中的Docker Node。

Docker Node的信息均来源于Swarm节点上注册的Docker Node。而一个Docker Node在Swarm节点上仅仅注册了Docker Node的IP地址以及Docker Daemon监听的端口号。使用swarm list命令时，需要指定discovery的类型，类型包括：token、etcd、file、zk以及<ip>。

15.4　总结

Swarm的架构以及命令并不复杂，本章初步介绍了Swarm、Swarm架构的设计与实现以及Swarm命令，希望能为学习Docker集群化管理的Docker爱好者带来帮助。

俗话说得好，没有一种一劳永逸的工具，有效地管理Docker集群同样也是如此。脱离场景来谈论Swarm的价值，意义并不会很大。相反，探索和挖掘Swarm的特点与功能，并为Docker集群的管理提供一种可选的方案，则是Docker爱好者更应该参与的事。

第16章　Machine架构设计与实现

16.1　引言

2014年12月，Docker官方在荷兰举办的DockerCon大会上宣布推出Docker Machine。此举预示着Docker从原先的单机部署迈向集群部署，Docker生态圈的能力更上一层楼。

Docker毕竟还是一个正在飞速成长的项目 ，尽管存在瑕疵，但是Docker在单机上的能力已经征服大量优秀的开发者。随着数十年来分布式系统的发展，分布式领域同样对Docker有着非常强烈的诉求。如何在服务器上自动化部署Docker，如何跨宿主机部署，如何使Docker的部署适应不同的基础设施环境，都会是Docker在分布式生产环境中落地时必须考虑的因素。

正是在如此背景下，Machine应运而生。Machine使得Docker的部署异常简易，不论是用户的单个主机，还是用户的数据中心，以及可能是第三方云平台提供商提供的云主机。Machine可以帮助用户在运行环境中创建虚拟机服务节点，在虚拟机中安装并配置Docker，最终帮助用户配置Docker Client，使得Docker Client有能力与虚拟机中的Docker建立通信。

一个大型的分布式环境中，Docker集群的从无到有，Machine甚至可以一键完成，这无疑将大大节省运维团队的人力、物力。同时，Machine完全有能力适应多种不同的底层基础设施，如OpenStack、VirtualBox、Amazon EC2、Azure、Rackspace、DigitalOcean、SoftLayer、Hyper-V等一系列国际知名基础设施提供平台。另外，通过Machine创建的虚拟机，不仅已经完成Docker的安装与配置，同时还可以保证环境中所有Docker配置的一致性。只要是通过Machine创建并部署的Docker节点，Machine就能对其进行远程管理或者执行Docker命令。

全球范围内，随着基础设施即服务(IaaS) 越来越完善以及Docker的逐渐成熟，大型分布式环境中Docker的集群化部署与配置，是一个必须攻克的问题。Machine作为Docker官方推荐的部署工具，深度关注与学习Machine将变得意义重大。

16.2　Machine架构

Machine可以帮助用户通过一条命令，从零开始，在极短的时间内，拥抱Docker。为实现此目标，用户需要准备一些前提条件，如安装Machine软件（docker-machine）、提供第三方的虚拟机、提供软件或基础设施平台。

Machine软件可以通过下载二进制文件获取，Docker官方的下载地址为：http://docs.docker.com/machine/。官方提供的版本可以支持三种操作系统，即Windows、OSX以及Linux。除此之外，用户可以通过编译Machine的源码生成Machine二进制文件，Machine的源代码完全托管于Github，地址为https://github.com/docker/machine。目前，Machine仍然处于Beta版本，功能以及特性都还在不断地变化中，因此Docker官方暂时还不建议将Machine运用于生产环境。Machine的版本更新非常迅速，本章以Machine v0.2.0为例，分析Machine的架构以及实现。

Machine的架构设计如图16-1所示。

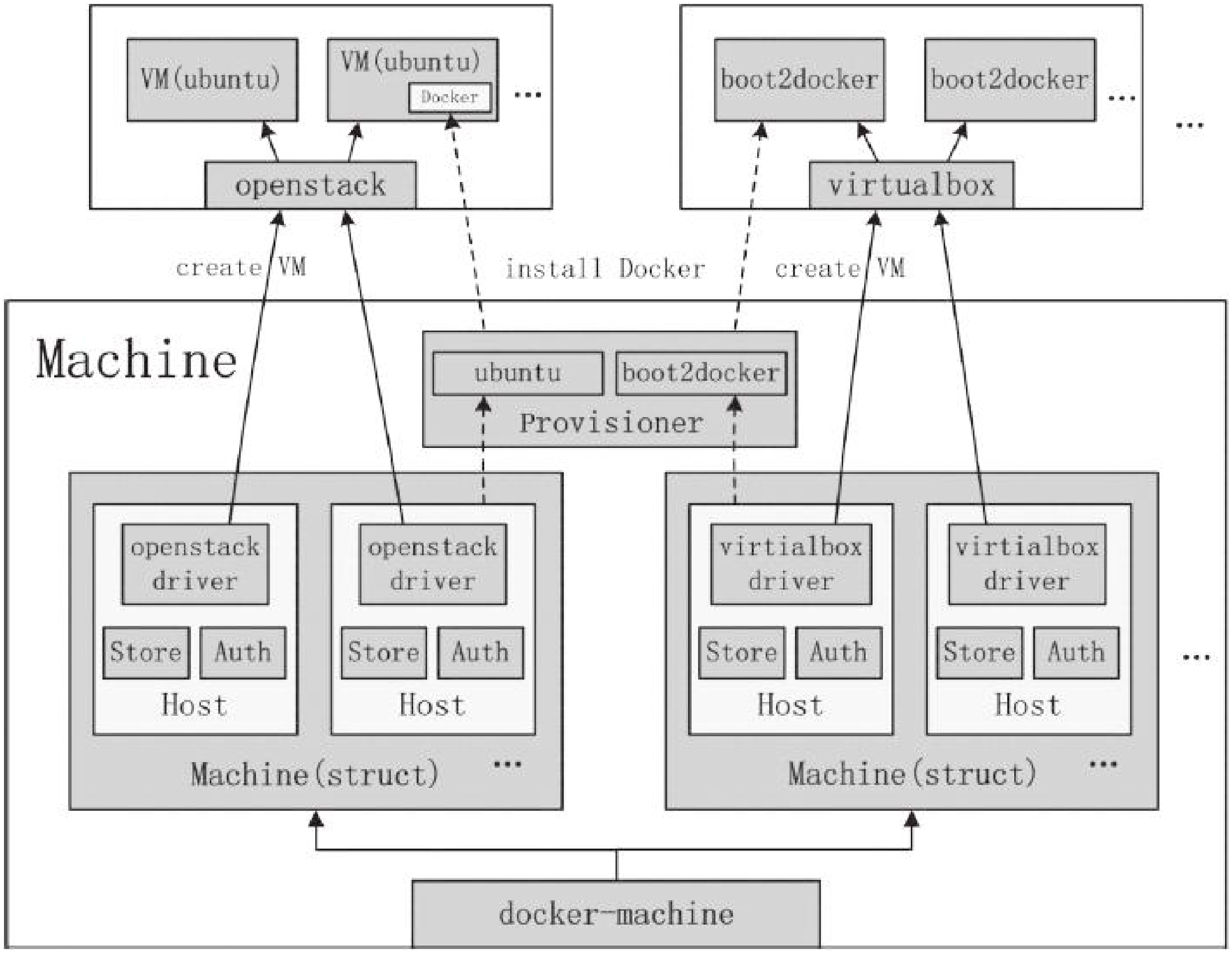


图16-1　Machine架构图

并非如Docker一般，Machine不是一个Client/Server架构。用户通过docker-machine命令使用Machine时，后台并非是一个常驻的服务进程。实际情况下，每次docker-machine命令被触发时，系统都会创建一个相应进程来完成用户指定的任务，任务完成后进程随即退出。由于Machine并没有后台进程，故通过内存来存储数据的方式也将行不通，此时Machine将所有创建的虚拟机信息以及Docker信息持久化至宿主机文件系统中。

为了更好地理解Machine的架构，首先我们来认识Machine中的几个重要的概念：Machine、Store、Host、Driver以及Provisioner。

16.2.1 Machine

此Machine并非代表Machine软件，也不指代docker-machine二进制文件，而是软件实现过程中抽象出的一个数据结构。

对于docker-machine create命令而言，一个Machine对象会创建，然后所有与该Machine相关的信息都存储到指定的路径中。

16.2.2　Store

对于每一个创建的Machine对象，相应的元数据都会被持久化到本地文件系统中，Store则用于告知用户这些元数据的存储位置。Store类型包含三个属性：root、caCertPath、privateKeyPath。属性root代表Machine对象信息存储的根目录；caCertPath代表连接Machine所需证书的路径；而privateKeyPath则代表通过ssh连接虚拟机时私钥所在路径。

16.2.3　Host

Host代表通过Machine在相应的基础设施上创建的虚拟机。每一个Host对象都包含Machine管理一台虚拟机所需要的所有信息。这些信息包括虚拟机名称（Name）、连接基础设施的driver名称（DriverName）、具体的Driver对象（Driver）、存储Machine实例所有信息的路径（StorePath）、包含Host具体信息的选项（HostOptions），另外还有与Swarm相关的信息以及认证内容。

值得一提的是HostOptions，其携带的虚拟机信息都在整个Machine中扮演重要角色。首先，包括虚拟机的内存大小、磁盘大小。另外，HostOptions的EngineOptions属性携带虚拟机中Docker Daemon的配置信息，包括：DNS配置、镜像存储driver、Daemon Label等众多配置信息；SwarmOptions属性代表是否为该虚拟机中创建一个Swarm节点；AuthOptions则是认证信息，用于建立用户与Docker Daemon之间的通信。抽象Machine源码，归纳总结出的Host数据结构如图16-2所示。

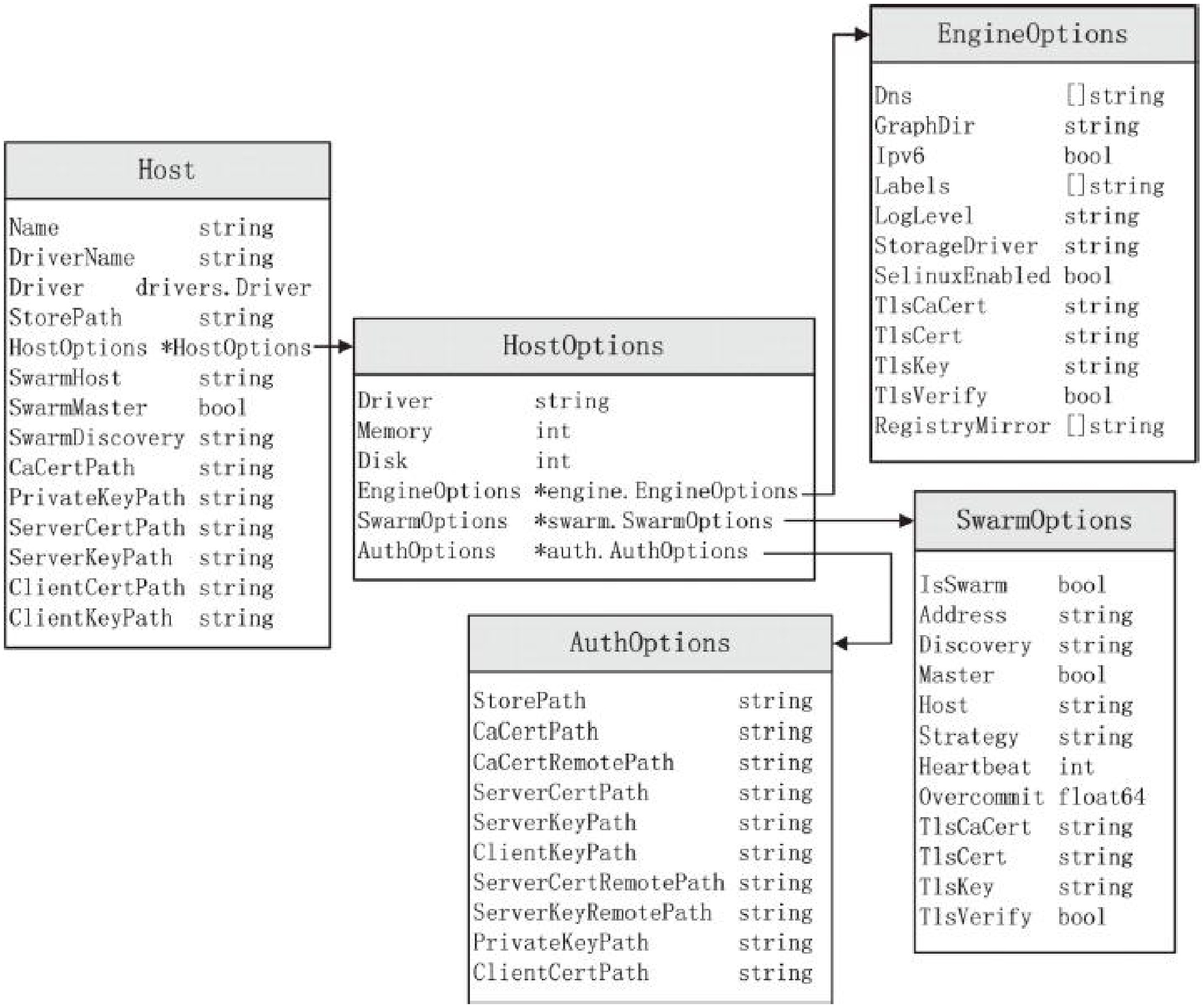


图16-2　Host数据结构

16.2.4 Driver

Driver在Machine中起到承前启后的作用，承前的是：所有创建虚拟机的请求，最终都需要落实到指定的Driver层；启后的是：Driver向具体的基础设施发起虚拟机创建请求，基础设施的响应也将首先返回至Driver。

16.1节提到，Machine目前支持众多的基础设施供应商。如此一来，由于不同基础设施平台之间存在对外接口的差异性，Machine很难做到提供一个大而全的Driver，涵盖所有基础设施，因此Machine中Driver的种类会与基础设施一一对应。事实的确如此，对于不同的基础设施，Machine使用不同的Driver来完成任务，openstack driver则完全适配OpenStack平台的接口，完成OpenStack平台上虚拟机的创建；对于VirtualBox这种虚拟化技术，Machine采用virtualbox driver，进而使用VBoxManage管理工具完成VirtualBox下虚拟机的申请……

16.2.5　Provisioner

Driver完成的工作是：在基础设施平台上从无到有创建虚拟机。然而，整个过程中并不涉及Docker、Docker Daemon的内容，换言之，创建虚拟机之后，虚拟机中并没有安装Docker。Provisioner对象则起到这最后一块拼图的作用。

Machine的使命就是在裸环境中创建装有Docker的机器，并统一化管理。因此，通过Driver创建虚拟机之后，Provisioner完成Docker安装与配置的任务。不同虚拟机有可能操作系统类型不同，而操作系统不同，Provisioner采取的手段也不同。目 前，Machine仅仅支持两种操作系统发行版的Docker安装，分别为Ubuntu和boot2docker。

一次完整的Provision流程，包含以下5个步骤：

1）为虚拟机设置主机名hostname；

2）若虚拟机中没有安装Docker，则为其安装Docker；

3）配置Docker Daemon参数，使其启动后仅仅接受TLS连接；

4） 复制证书至本地Machine实例的目录，并上传证书以及TLS认证信息至虚拟机；

5） 若用户指定Machine与Swarm的结合，通过Docker启动Swarm容器，并对其进行配置。

16.2.6 Machine运行流程

清楚Machine中重要的概念之后，我们一起进入Machine的运行流程一探究竟。回到图16-1的Machine架构图中，假设现在用户的需求是：在OpenStack平台上创建一台虚拟机，虚拟机的操作系统为Ubuntu，并且虚拟机中必须安装有Docker。我们以此为例，分析Machine的运行流程。

Machine的运行流程主要围绕Machine、Host、Provisioner展开，如下：

1）用户通过docker-machine运行用户命令。该命令中必须要包含openstack driver的名称，以及为所创建虚拟机指定的名称，命令的示例如下：docker-machine create-d openstack dev……

2）Machine解析Create命令，配置Store参数，检测driver类型，创建Machine对象，并配置hostOptions参数；

3）Machine通过虚拟机指定的名称，用户指定的driver，以及虚拟机配置信息hostOptions创建Host对象；

4）Machine将创建的虚拟机设置为active状态。

以上四个环节中，创建Host的环节涉及内容最多。Machine创建虚拟机首先需要获知driver类型，即用户需要在哪种基础设施平台上创建。解析-d参数，仅仅获取driver类型openstack还远远不够，用户还需要指定与openstack平台相关的诸多信息，如代表OpenStack认证URL的OS_AUTH_URL，OpenStack的用户名OS_USERNAME和密码OS_PASSWORD，OpenStack平台中镜像的ID“openstack-image-id”等。由于前面假设通过Machine创建Ubuntu虚拟机，故指定ID所代表的镜像应该为Ubuntu操作系统。

OpenStack driver解析出所有参数之后，通过driver内置的openstack client向基础设施平台发起虚拟创建请求。创建完毕之后，也是Provisioner登场之时。Provisioner的第一个任务便是获悉虚拟机内部的操作系统发行版类型（Ubuntu或者boot2docker），实现方式为：通过SSH远程登录虚拟机，并查看/etc/os-realease的内容，以此判断操作系统发行版的类型。类型的确定，意味着Provisioner可以对虚拟机实行有针对性的Docker安装计划，具体可以参见前面Provisioner的执行流程。

完成虚拟机的创建，完成Docker的安装，原则上来讲，创建工作已经完成。然而，为了让用户方便地使用新创建的Docker，Machine可以通过命令eval"$（docker-machine env dev）"，使

得本地的Docker Client无需显式指定即可连接远程的Docker Daemon。

16.3　Machine与Swarm的结合

Machine在Docker体系中的作用是：创建装有Docker的虚拟机。通过Machine强大的部署能力，我们并不能很好地调度和管理Docker集群。而Swarm的作用正是为Docker集群进行有效的管理与调度，若能将Machine与Swarm有效结合，必能为Docker集群的管理人员带来巨大的便利。

Machine的设计理念则充分考虑了这一点。只要用户有需求，Machine就完全有能力在虚拟机中安装Docker时，通过Docker Daemon启动一个Swarm Master节点，用来管理其余的Docker Node。当然，Machine也可以在安装Docker时，启动一个Swarm Agent容器，并将Docker Daemon作为一个Docker Node注册到指定的Swarm Master中，便于Swarm Master后续对于该Docker Node的调度。Swarm Master并未直接运行在虚拟机中，而是运行在Docker容器中。Machine与Swarm的关系如图16-3所示。

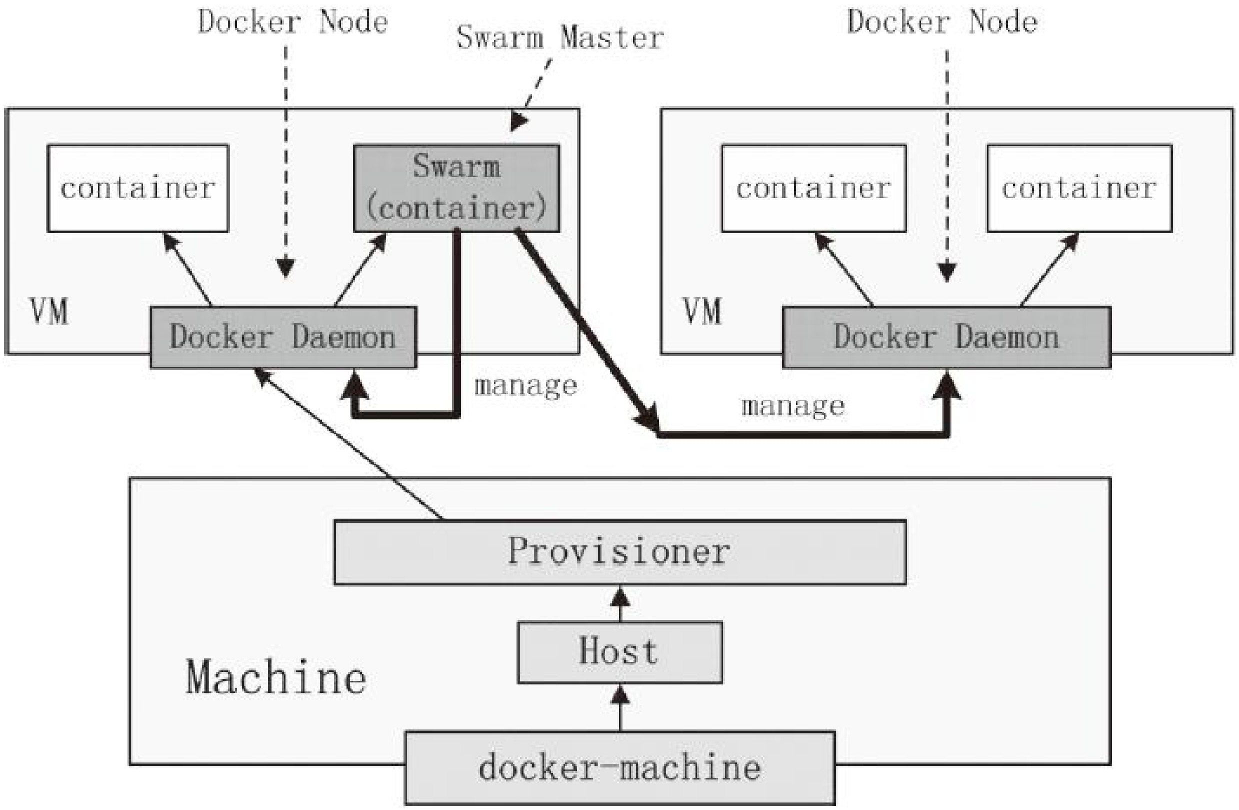


图16-3　Machine与Swarm的关系

关于如何在虚拟机上安装并配置Docker，16.2节已经讲述得较为清晰。事实上，Swarm的引入完全是在此基础上完成的。由于Swarm的运行环境为Docker容器，故Swarm的启动离不开Docker。实际运行过程也是如此，若用户指定安装Swarm，则Machine首先通过Provisioner在虚拟机中安装Docker，并在Provisioner安装与配置的最后一个步骤执行configureSwarm，完成Swarm的启动。

若需要配置Swarm，不论是Swarm Master还是Swarm Agent，用户均需在执行docker–machine create命令时传入相应的参数。

首先来看Swarm Master的配置流程。Docker安装完毕之后，Machine中的Porvisioner模块根据用户的需求，需要在虚拟机上配置Swarm Master，故Provisioner通过SSH连接至虚拟机中，并执行docker pull swarm：latest请求，用于下载swarm镜像。值得一提的是，swarm镜像不仅可以用来启动Swarm Master，同样可以用来启动Swarm Agent。由于用户指定启动Swarm Master，故Provisioner又执行docker run swarm：latest manage……命令，直接启动Swarm Master，并管理传入的Docker Node。

一般而言，一旦用户要求配置Swarm，不论配置Swarm Master还是Swarm Agent，虚拟机都会配置Swarm Agent，并受到Swarm Master的管理。如此一来，会出现Swarm Master与Swarm Agent处于同一个Docker Node上，而这种情况其实并不矛盾，Swarm Master也可以很好地管理自身所在的Docker Daemon。对于Swarm Agent的配置，Provisioner则执行docker run swarm：latest join……命令，使得Swarm Agent注册到指定的Swarm Master。

Swarm Master与Swarm Agent的配置完毕，则意味着Swarm集群的成功创建。Machine的存在很大程度上降低了用户与基础设施打交道的成本，从0到1快速创建Swarm集群。

16.4　总结

面对大型分布式环境或者尚未利用的基础设施，如何快速、有效、方便地使用Docker，已经不再是一个脱离现实的问题。某种程度上讲，Machine的诞生使得Docker技术的普及超越了开发者的单机模式，更是为中大型企业大规模运用Docker提供了可能性。

然而，对于不同的场景，Machine的价值也不尽相同，快速创建Docker并分发虚拟机的模式，在大规模的开发环境下，能为开发者带来很大的价值。Machine与Swarm甚至其他容器调度引擎的结合，更是为大规模集群的管理与调度提供了莫大的参考价值。另外，Machine对多基础设施的支持，同样大大降低了云平台迁移的难度，为混合云的发展，提供强有力的事实依据。

第17章　Compose架构设计与实现

17.1　引言

众所周知，随着不断地发展与完善，Docker的API接口变得越来越多。尤其在容器参数的配置方面，功能的完善势必造成参数列表的增长。若在Docker的范畴内管理容器，则唯一的途径是使用Docker Client。而Docker Client最原生的使用方式是：利用docker二进制文件发送命令行命令。一些特殊的应用场景下，容器管理过程的配置项极为冗长，甚至很可能是多容器的环境。因此，通过命令行来完成容器管理显然不是长久之计。很长一段时间内，全球的Docker爱好者都在探索以及寻找方便容器部署的途径。

Docker诞生于2013年3月，同年12月，基于Docker容器的部署工具Fig隆重登场。在Docker生态圈中，经过了两年多的洗礼，Fig项目得到飞速发展的同时，背后的东家也发生了很大的变化。作为Docker界容器自动化部署工具的翘楚，Fig原本是英国伦敦一家创业型公司的产品。随着产品的发展，Fig的巨大潜力受到工业界的普遍认可，在不到一年的时间内就受到Docker公司的密切关注。很快就在2014年7月双方爆出新闻：Docker收购Fig。收购完成之后，Fig改名为Compose，命令改为docker-compose。

17.2　Compose介绍

探听Fig与Compose的前世今生之后，让我们回到Compose本身，尝试挖掘Compose如何在Docker乱世中崭露头角，尝试分析Compose的技术以及定位又是如何。

认识一样新事物，从新事物的作用入手，往往不会出太大差错。而Compose最大的作用，就是帮助用户缓解甚至解决容器部署的复杂性。最原始的情况下，通过Docker Client发送容器管理请求，尤其是docker run命令，一旦参数数量骤增，通过命令行终端来配置容器较为耗时，同时容错性较差，且修复错误命令的时间成本很高。Compose则将所有容器参数通过精简的配置文件来存储，用户最终通过简短有效的docker-compose命令管理该配置文件，完成Docker容器的部署。

编辑配置文件与编辑命令行命令的难易程度高下立判。同时配置文件数据的结构化程度越高，可读性也会越强。传统情况下，如docker run等命令的参数数量很多时，由于flag参数的书写格式各异，很容易造成用户费解的情况；而配置文件中一行内容就是一类具体的参数值，可读性大大增强。

在生产环境下，Docker Client还有一方面经常被Docker爱好者所诟病，那就是难以进行多容器的管理，每次管理的容器对象最多只能是1个。容器虽然运行时相对非常独立，但是很多情况下，容器之间会存在逻辑关系，如容器A使用容器B的data volume，如容器C需要对容器D执行link操作等。对于有逻辑关联的容器，如果能将其作为一个整体，被工具统一化管理，那将大大减少用户的人为参与，提高部署效率。

诱人的功能与软件的完美之间，往往不能画等号，Compose同样如此。毕竟Compose的调用对象为Docker，故Docker的发展将直接影响到Compose的未来。Docker尚且还没有达到完美的地方，更遑论Compose，因此Docker官方并不建议Compose爱好者在生产环境中使用该工具。除此之外，Compose本身也存在一些缺陷，不熟悉其本质，自然也会深陷其中，难以脱身。

Compose软件的开发绝大部分通过Python语言完成，而本章的分析均基于Compose 1.2.0版本。

17.3 Compose架构

Docker生态圈中，Compose扮演的是部署工具的角色。用户使用Compose时，首先需要将部署意图通过配置文件的形式交给Compose。这样的配置需求包括：容器的服务名、容器镜像的build路径、容器运行环境的配置等。以下是一个较为简单的Compose配置文件。此配置文件定义了两个服务，名称分别为web以及db。服务web的镜像可以通过docker build来构建，Dockerfile所处目录为该配置文件当前目录；服务web需要对db服务进行link操作；最终服务web将宿主机上的8000端口映射到容器内部的8000端口。服务db通过镜像postgres来创建。

```
web:
  build: .
  links:
   - db
  ports:
   - "8000:8000"
db:
  image: postgres
```

配置文件在Compose体系中不可或缺。Fig时代支持的配置文件名为fig.yml以及fig.yaml；为了兼容遗留的Fig化配置，目前Compose支持的配置文件类型非常丰富，主要有以下5种：fig.yml、fig.yaml、docker-compose.yml、docker-compose.yaml以及用户指定的配置文件路径。

配置文件的存在为Compose提供了容器服务的配置信息，在此基础上，Compose通过不同的命令类型，将用户的docker-composer命令请求分发到不同的处理方法进行相应的处理。用户docker-compose的命令类型有很多，如命令请求docker-compose up……的类型为up请求，Compose将up请求分发至隶属于up的处理方法来处理；命令请求docker-compose run……的类型为run，Compose将run请求分发至隶属于run的处理方法来处理。

对于不同的docker-compose请求，Compose将调用不同的处理方法来处理。由于最终的处理必须落实到Docker Daemon对容器的部署与管理上，故Compose最终必须与Docker Daemon建立连接，并在该连接之上完成Docker的API请求。事实上，Compose借助docker-py来完成此任务。docker-py是一个使用Python开发并调用Docker DaemonAPI的Docker Client包。需要说明的是，毕竟docker-py作为Docker官方的一个Python软件包，和Docker并不隶属于同一个项目，因此docker-py在很多方面的发展均会滞后于Docker，即理论上而言，docker-py支持的API接口理论上会比Docker Daemon原生支持的API接口要少。因此，当使用docker-py作为Docker Client访问Docker Daemon时，确定API版本的支持是非常有必要的一步。另一方面，在docker-py支持的Docker API接口之中，Compose也并没有对其进行百分之百的实现，而这主要受限于Compose自身的软件定位。

清楚Compose的配置文件、处理方法以及docker-py概念之后，再来分析Compose的架构，如图17-1所示。

在Compose架构中，我们可以发现三个新的部分，分别为：project、service以及container。这三个概念均为Compose抽象的数据类型，其中project会包含service以及container。首先介绍这三者的意义。

project代表用户需要完成的一个项目。何为项目？Compose的一个配置文件可以解析为一个项目，即Compose通过分析指定配置文件，得出配置文件所需完成的所有容器管理与部署操作。例如：用户在当前目录下执行docker-compose up-d，配置文件为当前目录下的配置文件docker-compose.yml，命令请求类型为up，-d为命令参数，对于配置文件中的内容，Compose会将其解析为一个project。一个project拥有特定的名称，并且包含多个或一个service，同时还带有一个Docker Client。

Service，代表配置文件中的每一项服务。何为服务？即以容器为粒度，用户需要Compose所完成的任务。再次以本节前面配置文件为例，配置文件中共包含了两个service，第一个名为web，第二次名为db。一个service包含的内容，无非是用户对服务的定义。定义一个服务，可以为服务容器指定镜像，设定构建的Dockerfile，可以为其指定link的其他容器，还可以为其指定端口的映射等。

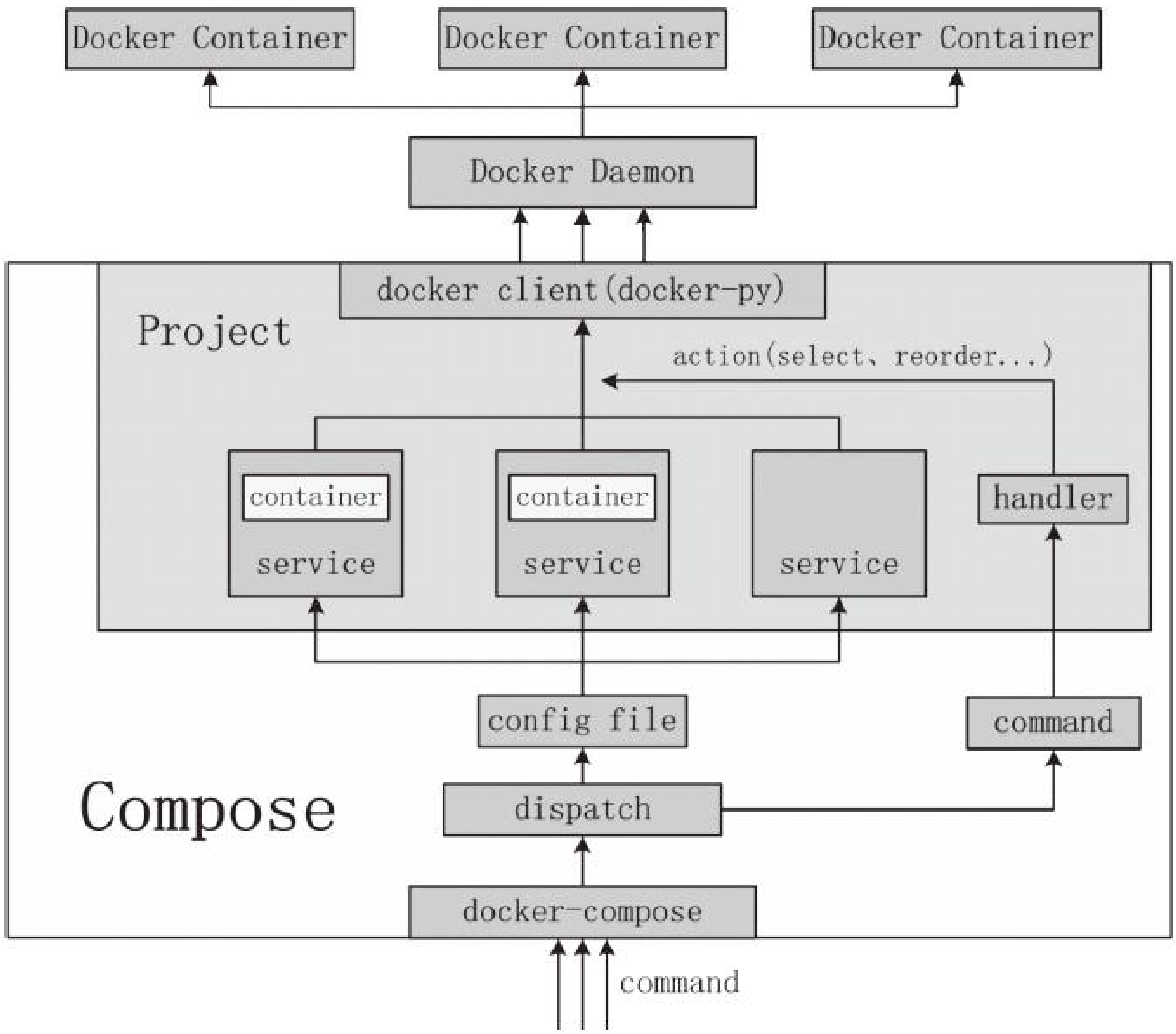


图17-1　Compose架构

从配置文件到service，实现了用户语义到Compose语义的转换。虽然一个service尽可能详细地描述了一个容器的具体信息，但是Compose并一定必须在service之上管理容器，如果用户使用docker-compose pull db命令，则仅仅完成db服务中指定镜像的下载。除此之外，Compose的service还可以映射到多个容器，如果用

户使用docker-compose scale web=3命令，则可以将web服务横向扩展到3个容器。

若用户对service的请求最终会落实到一个具体的容器上，则Compose会在service范畴内创建一个container对象，完成对具体容器的管理。container对象初始化时即集成了service的Docker Client，最终容器所有的操作，都由container对象通过调用Docker Client完成。

读到这里，大家可能会一个疑惑：Compose如何决策所有service的执行顺序？如果Compose一味按照配置文件中的书写顺序来完成service的指定任务，显然会出现一些不可避免的问题。假设多个service所描述的容器之间存在依赖关系，一旦配置文件中的顺序与实际的正常启动顺序不一致，必将导致容器启动失败。若在配置文件中容器A的描述位于容器B之前，而容器A的启动又依赖于容器B，此时若顺序执行A、B，A容器的启动必定将失败，而之后B容器可以正常启动。

一般而言，容器依赖关系会存在以下三种情况：存在links参数，容器的启动需要链接到另一个容器；存在volumes_from参数，容器的启动需要挂载另一个容器的data volume；存在net参数，容器的启动过程中网络模式采用other container模式，使用另一个容器的网络栈。为了解决这些问题，对于用户的某些请求，如docker-

compose up等，Compose在解析出所有service之后，需要根据各service的定义情况，梳理出所有的依赖关系，并最终以一个没有冲突的顺序启动所有的service容器。当然，这种情况无法应对环式依赖。

17.4　Compose评价

Compose的存在非常好地缓解了用户部署与管理容器的痛点。首先，Compose的存在使得用户无须再录入冗长的命令行命令，其次还扩宽了容器的部署范畴：从命令行的单容器部署到Compose的多容器部署。

17.4.1　Compose单机能力

Compose带来的益处显而易见，那是否意味着Compose可以适用于Docker容器部署的所有场景呢？答案自然是否定的。目前而言，Compose带来的是单机的便利，面对跨宿主机的容器部署与管理，Compose的能力依然薄弱。同时，Compose在单机上也并非可以完成所有的工作。

Compose的单机部署管理能力强大却并非完整。这主要体现在以下两方面。

首先在架构上，如前所述，Compose的能力会受限于两个方面：自身的设计理念以及docker-py对Docker API支持的完整程度。既然Compose定位为Docker容器的部署工具，那么只要自足于这个理念，它就不可能支持所有的Docker API接口，否则从功能上将会变得类似于第二个docker-py。然而，既然Compose中包装了service的概念，那么除了查看的service的状态之外，更为细致的情况下还应该能够查看service的资源使用情况。这个方面，Compose目前还没有集成，毕竟docker stats命令在Docker 1.5.0版本时才支持，Compose对这些API的支持显得有些滞后，docker-py的使用也是造成这类问题的重要原因。

其次在使用上，Compose的自动化部署并不意味着可以完全替代手动管理。还以容器依赖为例，虽然Compose目 前支持判断容器间的依赖，并生成合理无冲突的执行顺序，但是这样的执行顺序，仅仅在时序上有区分，并未在容器应用的逻辑上区分。例如，容器A为一个Web应用，依赖于容器B这个数据库，配置文件service A的links参数指明了service B，则Compose会在容器B启动完成之后再启动容器A。这看似合理，实则不然。对于容器A而言，应用程序需要创建到数据库的连接，一旦失败，它可能直接退出。而数据库容器B的启动需要一个过程，只要dockerinit进程开始执行，Docker Daemon均会认为该容器已经正常启动，故Compose通过Docker Client（ docker-py）也会认为容器B已经启动，从而立即启动容器A。由于此时容器B中的数据库应用仍需要完成初始化，直至最终数据库服务引擎完全启动之后，才能监听特定端口，接受外界应用的连接。B容器需要一定时间来完成这个阶段，而Docker Daemon不会理会容器内部的应用逻辑，直接认为容器启动完毕，最终Web应用A无法连接B中的数据库服务，自然会异常退出。总的来说，Compose还是立足于容器层的部署，并不涉及容器内部应用层的逻辑。而手动部署的方式，则可以在容器部署完毕时，人为测试容器内部应用程序的状态，从而以此为依据，确定是否部署受依赖的容器。

17.4.2　Compose跨节点能力

Docker容器跨主机部署的需求正逐步增大，稍令人失望的是，Compose目前在这方面的功能依旧不令人满意。实际应用场景下，Docker用户往往希望将不同类型的容器部署在不同的Docker节点上，满足负载、安全、资源利用等多方面的考虑。虽然Compose目前不具备这样的能力，但并不意味着Docker会放弃这方面的市场。

17.4.3　Compose与Swarm

Docker容器跨节点部署方案的发展，用“需求决定方向”来形容再准确不过。目 前，Docker正在酝酿着Compose与Swarm的深度结合，目 标是：使用户在一个Swarm集群上运行Compose来部署容器，效果和在单机上使用Compose完全一致。

先分析跨节点容器没有依赖的情况。容器之间一旦没有依赖，容器对自身所处的节点位置也就没有太多需求。这种情况下，理论上，Compose完全可以通过Swarm的label环境变量，将容器与满足条件的Docker Node联系在一起；同时也可以通过环境变量affinity，使几个容器部署在同一个Docker Node上或者避免在同一个Docker Node上。

再研究跨节点容器存在依赖的情况。跨节点容器有依赖，第一个需要解决的问题是跨节点容器的通信能力。而在Docker的范畴内，如果不借助其他工具，跨节点容器的通信目 前还没有很好的支持。因此，如links、volumes_from、net：container等容器依赖的情况，下一版本的Compose还会默认将相应的容器在同一个主机上部署运行。

17.5　总结

本章从Compose的历史与定位入手，随后简单分析架构设计与内部实现，最后评价了Compose在容器部署方面的能力。

Compose的诞生，并不是为了解决一切部署问题。如果信服这一点，那么研究Compose的应用场景就变得极为有价值。什么样的场景下，可以借助Compose发挥自动化部署的魅力，什么样的场景下，Compose并不能满足需求，而需要用户自行开发工具进行部署与管理，才是在Docker容器部署与管理领域值得深究的话题。Docker生态圈仍需要不断地发展，在容器缺乏部署工具方面，Compose仍然会处于一枝独秀的地位。清楚Compose的原理，对于Compose的运用定会有很大的帮助。